AUTOMATED UPGRADING OF COMPONENT-BASED APPLICATIONS

BY

DANIEL DIG

M.S., Politechnics University of Timisoara, 2002
B.S., Politechnics University of Timisoara, 2001

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois

Doctoral Committee:

Research Associate Professor Ralph Johnson, Chair
Faiman/Muroga Professor Marc Snir
Associate Professor Samuel Kamin
Assistant Professor Darko Marinov
David Bruton Jr. Centennial Professor Don Batory, University of Texas at Austin

Abstract

Software developers like to reuse software components such as libraries or frameworks because it lets them build a system more quickly, but then the system depends on the components that they reused. Ideally, the programming interface (API) to a component never changes. In practice, components change their APIs. Upgrading an application to the new API is error-prone, tedious, and disruptive to the development process. Although some tools and ideas have been proposed to solve the evolution of APIs, most upgrades are done manually. This makes maintaining software expensive. Our goal is to automate the upgrading and make it practical.

Our study of the API changes in five components revealed that over 80% of the changes that break existing applications are caused by refactorings. Refactorings are program transformations that improve the structure of existing components. We suggest that refactoring-based upgrading tools can be used to effectively upgrade applications. We propose an approach that is both automated and safe, without any overhead on the component producers. First, component refactorings are automatically detected (either inferred or recorded), then they are incorporated into applications by replaying. This is the process used to automate the upgrading of applications: along with the new version of the component, component developers ship the log of refactorings applied to create the new version. An application developer can then upgrade the application to the new version by using a refactoring tool to replay the log of refactorings.

We developed a toolset to automatically upgrade applications in response to component refactorings. First, we developed a record-and-replay extension to a refactoring engine which records component refactorings and replays them on the applications. To handle those cases when component refactorings are not recorded, we developed RefactoringCrawler, a tool that detects refactorings in Java components. The empirical evaluation of RefactoringCrawler shows that it scales to real-world components, and its accuracy in detecting refactorings is over 85%, a significant improvement over existing solutions.

Not only components evolve, but applications evolve too. Besides refactorings, components and applica-

tions evolve through edits. Refactorings and edits can interfere with each other thus impeding the replay. To address this, we developed MolhadoRef, the first software merging system that intelligently merges refactorings and edits from components and applications, therefore upgrading the applications. Experimental evaluation shows that MolhadoRef automatically resolves more merge conflicts than traditional text-based systems while producing fewer merge errors.

To address those cases when the source code of the application cannot be changed in response to component refactorings, we developed a tool, RefactoringBinaryAdapter (ReBA). ReBA automatically generates an adapter between component and application thus enabling old binary applications to run with the latest version of the component without requiring changes in the application.

Not only does our toolset reduce the burden of manual upgrades, but it will influence component designers as well. Without fear that they break the existing applications, designers will be bolder in the kind of changes they can make to their designs. Given this new found freedom, designers will continue to refactor the design of software components to make them easier to understand and use.

To Monika, my dear and beloved wife, and to our coming child.

Acknowledgments

During the writing of this dissertation I have learned from many people. Without their input and support, this dissertation would not be what it is.

First, I want to thank my adviser and mentor, Ralph Johnson. Ralph is my role model for making a difference in the practice of software development. His work has had a tremendous impact during the last two decades of software engineering: design patterns and refactoring are continuing to influence the lives of millions of software developers. He is the most pragmatic academic that I have ever met; he has taught me to value the things that are really important. Ralph is the one that seeded the idea of record-and-replay of refactorings during our first meeting in 2002, but he also gave me the freedom to explore the other ideas in this dissertation, being my constant supporter. He believed in me and took me along to conferences, even before I started to publish. I also extend many thanks to my PhD committee members, Darko Marinov, Sam Kamin, Marc Snir and Don Batory who offered guidance and support. Darko's contagious passion for research flamed my own desire to publish. The idea of inferring refactorings in addition to recording them came out during one of our long technical discussions; I always left invigorated after these talks. He has been a careful reviewer of all my writings, always asking the tough questions. I hope that our friendship and collaboration will continue to flourish in the years to come. Sam is the one who always asked me insightful questions at my practice talks. I also learned so much from the insights he often brought in during our PL reading group. Marc granted me an introduction to the wonderful world of parallel programming. He always surprises me with his keen ability to grasp any area of computer science. I am especially thankful to him for supporting me for two semesters through a TA/RA position. Don is the external member of my committee from UT Austin. He has an elegant algebra of program transformations, which we discovered can nicely describe some of my dissertation research. I learned so much from the long discussions we had at conferences, and from several discussions through e-mail. I look forward to collaborating in the future.

Tien Nguyen from Iowa State University has been a long time collaborator. His expertise in Software

Configuration Management nicely complemented my own lack of expertise in that area. In addition to technical discussions, we have had wonderful chats about the academic life.

I have been lucky to pair-program with some very talented master's students. They were my constant companions in bouncing ideas back and forth and they helped a lot with the implementation of the tools presented in this dissertation. Can Comertoglu was my pair-programming partner for RefactoringCrawler; Kashif Manzoor for MolhadoRef; Stas Negara and Vibhu Mohindra for ReBA. I learned as much from them as they learned from me. It was my great joy to see each of them grow and mature.

I have learned from many discussions with my colleagues, from their comments on various drafts of papers that make up this dissertation, and from their feedback during practice talks. I am thankful to Paul Adamczyk, Brian Foote, John Brant, Federico Balaguer, Alejandra Garrido, Joe Yoder, Brian Marick, Jeff Overbey, Munawar Hafiz, Marcelo d'Amorim, Steve Lauterburg, Brett Daniel, Nicholas Chen, Maurice Rabb, Roger Whitney, Tanya Crenshaw, Weerasak Witthawa. I am especially thankful to Paul Adamczyk for being my chief reviewer. He has a special gift for noticing errors that nobody else would notice. Brian Foote was extremely helpful in playing the devil's advocate. I knew that if my talk would pass Brian, it would surely satisfy any other audience. John Brant was introduced to me as the smartest programmer that Ralph Johnson knew. To this day he maintains his reputation; he always amazes me with his insights.

For insightful comments that improved the presentation of my scientific papers, I am grateful to Frank Tip, Adam Kiezun, Russ Rufer, Mathieu Verbaere, Steve Berczuk, Johannes Henkel, Amer Diwan, Oscar Nierstrasz, Erich Gamma, Stéphane Ducasse, Serge Demeyer, Filip Van Rysselberghe, Doru Girba, Riley White, Danny Soroker, Bob Hanmer, and anonymous reviewers. I received valuable feedback from several others during conversations at conferences: Martin Fowler, Kent Beck, Joshua Kerievsky, Ward Cunningham, Dick Gabriel, Linda Rising, Dragos Manolescu. Despite my best intentions, it is possible that I might have omitted someone from these lists. Please accept my apologies and be assured that I am grateful for your help.

Thanks to the University of Illinois Graduate College for awarding me a Dissertation Completion Fellowship, providing me with the financial means to complete this project. I want to thank the Department of Computer Science at the University of Illinois for awarding me an Outstanding Mentoring Fellowship and for giving me a faculty office during my last months of work in Urbana. I am very grateful to IBM for an Eclipse Innovation Grant that supported me during the most critical years. I am thankful to Agile Alliance

for a Travel Grant that made it possible to collaborate with researchers from Iowa State University.

Monika, my dear and beloved wife, has been my greatest supporter, and my best friend for almost a decade. I am constantly learning from her how to serve others selflessly, how to be humble, and how to deeply care about those around us. I do not have enough words to express my gratitude for all that she means to me.

And finally, I want to thank God for granting me the wisdom and the grace to work on these projects. He has brought me “out of nowhere” into an environment where I could grow and learn every day. His gift of salvation is the most precious treasure in my life.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Component API Changes Create Problems for Applications

Most programmers do not build applications from scratch. Instead, they build on top of existing software components (e.g., libraries and frameworks). For example, graphical software applications are built by reusing libraries of graphical widgets (e.g., windows, menus, and scroll bars). Software components expose a set of *Application Programming Interfaces (APIs)* that software applications use to interact with such components.

Ideally, the interface to a component never changes. In practice, new versions of components often change their interface. For example, between two major versions (one year apart) of Eclipse [Ecla], a mature and very popular framework for developing Integrated Development Environments (IDEs), there were 51 API changes that are not backwards-compatible. In Struts [Str], a popular framework for developing web applications, there were 136 API changes [DJ05, DJ06] not backwards-compatible between two major versions. Such API changes require applications that use the components to be changed (upgraded) before the new versions can be used. Traditionally, developers upgrade applications manually, which is error-prone, tedious, and disruptive in the middle of software development.

Moreover, upgrading to use new component versions is an important activity even after the software product has been released. New versions of software components contain performance improvements, bug fixes, and new features. Some component producers release versions frequently, thus API changes are few per each release, while others release infrequently, but with many API changes. Some organizations prefer to frequently upgrade to the latest versions of components, while others upgrade infrequently, but the cost of upgrade is higher. Because of lack of tool support, upgrading makes maintaining software expensive.

An important kind of change to object-oriented components is a refactoring [OJ90, FBB+99]. Refactorings are program transformations that change the structure of a program but not its behavior. Example

refactorings include changing the names of classes and methods to better reveal their intent, moving methods and fields between classes to improve cohesion and minimize coupling, extracting duplicated code into one single function to reduce duplication.

Refactoring engines are tools that automate the application of refactorings: first the programmer chooses a refactoring to apply, then the engine checks if the change is safe, and if so, transforms the program. Refactoring engines are a key ingredient of modern IDEs, and programmers rely on them to perform refactorings.

Refactoring engines can apply the refactorings to change the source code of a component. However, the refactoring engine operates within a *closed-world* paradigm: it can change only the source code that it has access to. Component developers often do not have access to the source code of all the applications that reuse the components. Therefore, refactorings that component developers perform preserve the behavior of the component but not of the applications that use the component; although the change is a refactoring from the component developers' point of view, it is not a refactoring from the application developers' point of view. There is a mismatch between the *closed-world* paradigm of refactoring engines and the *open-world* paradigm where components are developed at one site and reused all over the world. When refactorings change the component APIs, applications need to change accordingly in order to reuse the latest version of the component.

Tool support for upgrading applications has been a research topic for some time. Several authors [CN96, KH98, RH02, BTF05] propose that component developers write annotations that can be used by tools to upgrade applications. However, writing such annotations is cumbersome. Informal interviews with component developers [Dan] reveal that they are not likely to write such annotations. A more appealing approach would be if tools could generate this information.

1.2 Goal Statement and Research Questions

Our goal is to *reduce the burden of component reuse* by reducing the cost of adapting to change. Specifically, we want to relax the closed-world constraint of refactoring engines. We hypothesize that refactoring plays an important part during component API evolution. Our thesis is that *with respect to API changes caused by refactorings, it is possible to make the upgrading automated and practical*.

To be practical, our solution must address the needs of both component and application developers:

application developers want an automated and safe (i.e., behavior-preserving) way to upgrade component-based applications to use the newer versions of components; component developers are reluctant to learn any new language or write any specifications extraneous to the regular component development.

Our solution to upgrading applications stems from our belief that *software development (and hence, component development) is a sequence of program transformations*: developers take version N and transform it into version N+1. What are the program transformations that occur most often in practice? How do they affect clients of the component? During component evolution, most changes would add new APIs and features that do not affect old applications. However, to make the component more *additive* (e.g., to make it easy to add new APIs), developers need to constantly refactor the structure of the component. It is these changes that create problems for old applications.

Our solution to upgrading applications when their components change due to refactorings is to automatically recover component refactorings and then to automatically incorporate them on the applications. Along with the new version of the component, the component developers ship the log of refactorings applied to create the new version. An application developer can then upgrade the application to the new version by using a tool that safely incorporates this log of refactorings.

One such possible solution is *record-and-replay*: an extended refactoring engine automatically records API refactorings applied on the component. At the application site, the tool automatically replays these refactorings in the enlarged context of old component and application. Now, because the refactoring engine has access to both the code of the component and application, replaying the component refactorings changes also the application. Although this solution is elegant from an engineering standpoint, there are further questions that need investigation and other tools needed to overcome the limitations of record-and-replay. For example, recorded refactorings are not available for the current or legacy versions of components, some component refactorings are performed manually while others might not be applicable in the context of the application, or the application could not be changed due to lack of source code.

In this dissertation, we investigate the following key questions regarding automated upgrading:

1. How much of the component evolution can be expressed in terms of refactorings? Do refactorings carry both the syntax and the semantics of component changes?

2. Can refactorings be gathered automatically or with minimal involvement from component developers? Is the accuracy of the detection good enough to be used in practical applications?

3. Can component refactorings be incorporated automatically and safely in applications? How do component refactorings interfere with other changes made at the application side?

4. What if the source code of the application is not available and/or cannot be changed in response to component refactorings? Can applications be shielded from component refactorings such that the old application could work with the new version of the component without recompilation?

To answer these questions, we conducted several investigations. The first one characterizes changes in real-world components [DJ05, DJ06]. The next ones [DCMJ06, DMJN07a, DMJN07b, DNJM07] build and evaluate a tool-set that automates upgrades. This tool-set comprises an extension to the Eclipse refactoring engine to record-and-replay refactorings, followed by RefactoringCrawler, a tool to automatically infer refactorings, then MolhadoRef, a tool to merge component and application refactorings and API edits, and ReBA, a tool to automatically generate a compatibility wrapper that allows old applications and new versions of the component to "talk" the same language.

1.3 Approach to Automatically Upgrade Applications

1.3.1 The Role of Refactorings in API Evolution

To learn what types of API changes cause problems for application developers, we looked at four well known frameworks and libraries from the open source realm (Eclipse, Struts, JHotDraw, and log4j) and one proprietary framework (e-Mortgage) from a large bank.

We discovered that the changes that break existing applications are not random, but they tend to fall into particular categories. Over 80% of these changes could be considered *refactorings* if looked at from the point of view of the component itself [DJ05, DJ06]. Since refactoring plays such an important role as mature components evolve, we propose that refactorings are used to formally express component evolution.

Probably the most important advantage of expressing component evolution in terms of refactoring operations is that refactorings carry deep semantics of the change along with the syntax. Their semantics are discussed in refactoring catalogs [FBB$^+$99] and are incorporated in refactoring engines.

Semantics become crucially important in OO languages where programmers make heavy use of class inheritance and dynamic dispatch of method calls at runtime. For instance, a simple refactoring like renaming a public method in a class involves not only the renaming of the method declaration, but also updating all

its call sites, renaming all methods in the class hierarchy that are overridden by or that override the method under discussion, as well as updating all the call sites of these methods. Before applying such an operation, the refactoring engine checks whether the new name would result in a name collision or in a potential change in method dispatch. The thorough analysis ensures that the change preserves the existing behavior of the software, thus leading to safe updates.

1.3.2 Record-and-replay of Refactorings

A small extension to the refactoring engine allows it to record the refactorings when they are applied on the component. A log of these refactorings can be shipped along with the new version of the component. At the application's site, this log of refactorings is replayed with the extended refactoring engine on both the older version of the component and the application. When applying each refactoring on the component, the refactoring engine can now find and update all the references to the refactored program elements that belong to the application. When replaying a sequence of refactorings, it behaves like a transaction: either all refactorings are applied, or no refactoring is applied.

Record-and-replay of refactorings was recently demonstrated in CatchUp [HD05], JBuilder2005, and Eclipse 3.2 (we collaborated on this extension of Eclipse).

1.3.3 Automated Detection of Refactorings

While replay of refactorings shows great promise, it relies on the existence of refactoring logs. However, logs are not always available for existing versions of components. Also, logs will not be available for all future versions; some developers will use refactoring engines that do not record, and some developers will perform refactorings manually. To exploit the full potential of replay of refactorings, it is therefore important to be able to infer them.

We propose a novel algorithm that detects refactorings applied between two versions of a component. Our algorithm works in the realistic open-world paradigm where components are reused outside the developing organization. Interface changes do not happen overnight but follow a long *deprecate-replace-remove* lifecycle. Obsolete entities (marked as deprecate) will coexist with their newer counterparts until they are no longer supported. Also, multiple refactorings can happen to the same entity or related entities. This style of development introduces enough challenges that existing algorithms [DDN00, RD03, APM04, GZ05, WD06]

for detection of refactorings cannot accurately detect them. In addition, real-world software components can be large, hundreds of thousands of lines-of-code (LOC). Syntactic analyses are too unreliable, and semantic analyses are too slow.

Our algorithm overcomes all these challenges. We implemented our algorithm in an Eclipse plugin, called RefactoringCrawler, that works for Java components. To achieve both high accuracy and scalability, RefactoringCrawler combines a fast syntactic analysis to detect refactoring candidates and a precise semantic analysis to refine the results. Our syntactic analysis is based on Shingles encoding [Bro97], a technique from Information Retrieval. Shingles are a fast technique to find similar fragments in text files; our algorithm applies shingles to source files. Most refactorings involve repartitioning of the source files, which results in similar fragments of source text between different versions of a component. Our semantic analysis is based on *reference graphs* that represent references among source-level entities, e.g., calls among methods[1]. This analysis considers the semantic relationship between candidate entities to determine whether they represent a refactoring.

RefactoringCrawler currently detects seven types of refactorings, focusing on those refactorings that appear most often in practice [DJ06]. We have evaluated RefactoringCrawler on three components ranging in size from 17 KLOC to 352 KLOC. The results show that RefactoringCrawler scales to real-world components, and its accuracy (both precision and recall) in detecting refactorings is over 85%.

1.3.4 Automated Replay of Refactorings

During the replay, the upgrading tool needs to handle those cases when a refactoring cannot be played back in the new context. For instance, at the component site it was possible to rename an API method, but at the application site this renaming results in a conflict if the application already defined a method with the new name. Currently, refactoring engines quit the operation if such a conflict arises.

In addition, besides refactorings, components evolve through edits. Although it is currently infeasible to automatically recover the semantics of all edits, those at the API level have well defined semantics. For example, when adding a new method, there should not be any other method with the same name and signature in the same class. Moreover, because both the component and application evolve simultaneously, upgrading the application effectively means *merging* component and application changes.

[1]These *references* do not refer to pointers between objects but to references among the source-code entities in each version of the component.

We developed an algorithm and a tool, MolhadoRef, that semantically merges refactorings and API edits from component and application, while low-level edits are merged syntactically. MolhadoRef uses the *operation-based* approach [LvO92] to record the change operations and replay them. However, replaying operations induces *dependence* relationships between them. The merge algorithm uses *inverted refactorings* [DMJN07a] to eliminate the dependences between refactorings and edits, and topologically sorts the refactorings so that they can be replayed. In addition, the merge algorithm uses refactoring-to-refactoring transformations when topological sort cannot solve all dependences.

MolhadoRef is aware of program entities and the operations that change them. Therefore, MolhadoRef has a superior performance when compared with traditional text-based merging algorithms (e.g., used in CVS). We compared MolhadoRef and CVS using a case study and controlled experiments. The results show that MolhadoRef automatically resolves more merge conflicts, after merging there are less compile errors and fewer runtime errors, while the total time to merge (including the human time) is much shorter.

1.3.5 Adapters to Rescue from API Evolution

Previous tools in our tool-set assume that the application source code can be changed in response to component refactorings. What if the application cannot be changed (e.g., because of not having access to the source code), or the application is not available at all? Even more, for legal reasons, some software licenses prevent even changes to the bytecodes of applications.

As an alternative to changing the application source code to accommodate changes in the components, a compatibility adapter can insulate the application from the component API changes. An adapter can be applied to the whole component such that all applications can still interact with the old APIs, or the adapter can be applied to a concrete application, such that it insulates only the specific APIs that are used by the application.

We developed a tool, ReBA, that automatically generates such a backward-compatible adapter from the trace of component refactorings and API edits. Executing an application with such an adapter adds some performance overhead (both more memory consumption and increased CPU cycles), but our evaluation on case studies and controlled experiments shows that the performance degradation is acceptable. Ultimately, the application developers will need to migrate to the new APIs using our tool-set; meanwhile, the generated wrappers are a good short-term compromise.

1.4 Contributions

This dissertation makes several important contributions. Some contributions bring significant improvements over the state-of-the-art tools for upgrading applications, while others break new ground in areas never explored before.

1. **Problem description.** This dissertation describes an important problem, the evolution of APIs, that appears in the practice of component-based software development.

2. **Insights into the evolution of APIs.** To the best of our knowledge, ours is the first quantitative and qualitative study that offers insights into the evolution of APIs of real-world software components. While others focused on *why* software components change, we focus on *how* they change.

3. **A rigorous approach and a theoretical framework.** We express the evolution of component APIs as refactorings, a set of program transformations with well defined semantics. The state-of-the-practice relies on textual descriptions in release change logs or on the textual differences between the source code of two component versions.

4. **A practical upgrading tool-set.** We develop a tool-set to automate the upgrading of component-based applications that is safe for application developers and practical (no overhead on component producers). RefactoringCrawler is significantly more accurate than previous state-of-the-art tools for detecting refactorings. MolhadoRef is the first tool that merges refactorings and edits intelligently. ReBA is the first tool that generates a compatibility adapter to insulate applications from component refactorings.

5. **Empirical evaluation.** We evaluate the tools in the tool-set through controlled experiments and case studies.

1.5 Organization

The rest of this dissertation is organized as follows: Chapter 2 presents our study to characterize the API changes in five existing software components. The first part presents several techniques that component developers use to preserve backward compatibility. The second part is a catalog of the API-breaking changes

that we witnessed in the five case studies. This study reveals that refactorings play an important role in the evolution of mature components: over 80% of the API changes that are not backwards compatible are caused by refactorings.

Chapter 3 presents background information on refactoring transformations and a short description of the design of the record-and-replay refactoring engine. It concludes with the main limitations of the record-and-replay and it shows how the other chapters address these limitations.

Chapter 4 presents the design and evaluation of RefactoringCrawler, our tool for detecting component refactorings. RefactoringCrawler scales to analyze real world components (half a million lines of code) and its accuracy in detecting refactorings is over 85%.

Chapter 5 rephrases the problem of incorporating component refactorings and API edits into applications as a software merging problem: the component development is viewed as one branch of development, application development is the other branch. Upgrading an application effectively means merging the changes in the two branches. MolhadoRef, our refactoring-aware merging tool, is far more effective than the state-of-the-practice text-based merging tools.

Chapter 6 presents the design and evaluation of ReBA, our tool for generating compatibility adapters. ReBA enables an older application to continue to run with the new version of a component, without requiring any changes to the application. Evaluation shows that the performance overhead imposed by our adapters is acceptable.

Chapter 7 surveys related work and Chapter 8 concludes and talks about future work.

Parts of this dissertation have been published in technical reports, conferences, and journals. In particular, Chapter 2 is described in an ICSM-2005 [DJ05] and JSME journal paper [DJ06], Chapter 4 in an ECOOP-2006 paper [DCMJ06], Chapter 5 in an ICSE-2007 paper [DMJN07a] and a TSE journal paper [DMJN07b], and Chapter 6 is currently under review [DNJM07] at a premiere software engineering conference. These chapters have been extended and revised when writing this dissertation.

Chapter 2

The Role of Refactorings in API Evolution

2.1 Introduction

Software evolution has long been a topic of study [LS80]. Others [CHK+01, MB98] have focused on *why* software changes; we want to discover *how* it changes so that we can *reduce the burden of component reuse* on applications.

Although there are principles of software evolution that are true for software in any language, programming languages have an impact on software evolution. We are particularly interested in the evolution of object-oriented components (we refer to both libraries and frameworks as components, unless a distinction is necessary). Classes contain a mixture of private and public methods. Public methods are meant to be used by application programmers. The set of public methods of a class library make up its API (Application Programmer Interface). Changes to private methods and classes do not pose a problem to application developers; they only care about changes to the API.

An important kind of change to object-oriented software is a refactoring [OJ90, FBB+99]. Refactorings are program transformations that change the structure of a program but not its behavior. The original work on refactoring was motivated by framework evolution. Opdyke [OJ90] looked at the Choices operating system and the kind of refactorings that occurred as it evolved. Graver [Gra92b] studied an object-oriented compiler framework as it went through three iterations. Tokuda and Batory [TB01] describe the evolution of two frameworks, focusing on how large architectural changes can be accomplished by a sequence of refactorings.

However, none of these studies determined the fraction of changes that are refactorings. Of the changes that cause problems for application maintainers, what fraction are refactorings? Are refactorings as important in practice as these authors imply? How much of the component evolution can be expressed in terms of refactorings? The only way to tell is to look at changes in a component over time and categorize them.

	Eclipse 3.0	e-Mortgage	Struts 1.2.4	log4j 1.3	JHotDraw 5.0
Size [KLOC]	1,923	52	97	62	14
API Classes	2,579	174	435	349	134
BreakingChanges	51	11	136	38	58
ReleaseNotes [Pages]	24	-	16	4	3

Table 2.1: Size of the studied components. The number of classes in API denote only those classes that are meant to be reused. ReleaseNotes give the size (in pages) of the documents describing the API changes. The logs were provided by the component developers.

We looked at four frameworks and one library (see Table 2.1 and Section) developed by five different groups. Four are commonly used open source and one is a proprietary framework. All the case studies are mature software, namely components that have been in production for more than three years. By now they have proved themselves to be useful and therefore acquired a large customer base. At this stage, API changes have the potential to break compatibility with many older applications. Backwards compatibility and different strategies to preserve it are the topic of Section 2.3.

We analyze and classify the API changes in the five systems. We describe a few techniques used to maintain backwards compatibility (Section 2.3.2) while the main focus is on the API changes that break compatibility with older applications (Section 2.4). We learned that between two versions for each of the five systems we studied, respectively 84%, 81%, 90%, 97%, and 94% of the API breaking changes are the result of refactorings. Most API changes occur as responsibility is shifted between classes (e.g., methods or fields moved around) and collaboration protocol changes (e.g., renaming or changing method signature). These results made us believe that refactoring plays an important role as mature components evolve.

2.2 Study Setup

We chose four well known frameworks and libraries from the open source realm. To check whether the production environment affects the type of API changes, we chose one more proprietary framework. We tried to be unbiased in the selection of the case studies, the main concern being that the systems have good documentation. Subsection 2.2.1 describes briefly the components that we used as case studies.

For each component we chose for comparison two major releases that span large architectural changes. There are two benefits to choosing major releases as comparison points. First, it is likely that there will be lots of changes between the two versions. Second, it is likely that those changes will be documented thus

providing some starting point for a detailed analysis of the API changes. Subsection 2.2.2 describes the process we followed to collect and analyze these API changes.

2.2.1 Overview of the Case Studies

Eclipse Platform

Eclipse [Ecla] was initially developed by IBM and later released to the open source community. The Eclipse Platform provides many APIs and many different smaller frameworks. The key framework in Eclipse is a plug-in based framework that can be used to develop and integrate software tools. This framework is often used to develop Integrated Development Environments (IDEs). The Eclipse Platform is written in Java.

We chose two major releases of Eclipse, namely 2.1 (March 2003) and 3.0 (June 2004). Eclipse 3.0 came with some major themes that affected the APIs. The *responsiveness* theme ensured that more operations run in the background without blocking the user. New APIs allow long-running operations like project builds and searches to be performed in the background while the user continues to work.

Another major theme in 3.0 is *rich-client platforms*. Eclipse was designed as a universal IDE. However many components of Eclipse are not particularly specific to IDEs and can be reused in other rich-client applications (e.g. plug-ins, help system, update manager, window-based GUIs). This architectural theme involved factoring out non-IDE specific elements. APIs heavily affected by this change are those that use the filesystem resources. For example, `IWorkbenchPage` interface is used to open an editor for a file input. All methods that were resource specific (those that dealt with opening editors over files) were removed from the interface. A client who opens an editor for a file should first convert the file to a generic editor input. Now the `IWorkbenchPage` interface can be used by both non-IDE clients (e.g. an electronic mail client that edits the message body) as well as IDE clients.

e-Mortgage Framework

A large banking corporation in the Midwest has been building an electronic mortgage framework, e-Mortgage (closed-source, proprietary software), to leverage existing financial expertise when writing new applications.

The e-Mortgage framework allows various banking applications developed within the company to communicate with each other and with the existing legacy systems. The framework receives requests from front-end systems or services, evaluates their requirements and redirects the request to a specific destina-

tion, or destinations such as a pricing engine or closing cost engine. After receiving an appropriate response, the framework refines it for a specific request channel and then forwards it back to the requestor.

When we visited the banking institution, they were finalizing the integration between the mortgage framework and another middleware framework developed independently at another branch of the bank. Frameworks are designed for extension not for integration [MBF99]. As a result of the marriage between the two frameworks, the application developers had to migrate the existing services. The company reported that the whole integration and upgrading process lasted a summer. At the time we write this, there are about 50 services that use the framework.

Struts Framework

Struts [Str] is an open source framework for building Java web applications. The framework is a variation of the Model-View-Controller (MVC) design paradigm. Struts provides its own Controller component and integrates with other technologies to provide the Model and the View.

For the Model, Struts can interact with standard data access technologies, like JDBC and EJB, as well as most any third-party packages, like Hibernate, iBATIS, or Object Relational Bridge. For the View, Struts works well with JavaServer Pages, as well as Velocity Templates, XSLT, and other presentation systems. Because of this separation of concerns, Struts can help control change in a Web project and promote job specialization.

We chose for comparison version 1.1(June 2003), a major past release, and 1.2.4 (September 2004), another stable release. Many API changes reveal consolidation work: the framework developers eliminated duplicated code and removed unmaintained code or code with high defect-rates.

log4j Library

log4j [Log] is a popular Java library for enabling logging without modifying the application binary. It allows the developer to control which log statements are output with arbitrary granularity by using external configuration files. Logging does have its drawbacks. It can slow down an application. If too verbose, it can cause scrolling blindness. To alleviate these concerns, log4j is designed to be reliable, fast and extensible.

log4j uses a logger hierarchy to control which log statements are output. This helps reduce the volume of logged output and minimize the cost of logging. The target of the log output can be a file, an OutputStream,

a java.io.Writer, a remote log4j server or a remote Unix Syslog daemon logger among many other output targets.

We chose for comparison version 1.2 (May 2002) and version 1.3alpha6 (January 2005). The library passed through an expansionary phase and it grew from 30KLOC to 62KLOC. The library grew by improving on existing components (like Chainsaw, a visualization toolkit for loggers) or adding new components (like support for plugins as a way to extend the library).

JHotDraw

JHotDraw [JHoa] is a 2D graphics framework for structured drawing editors. It is written in Java although it was originally developed in Smalltalk by Kent Beck and Ward Cunningham. Erich Gamma and Thomas Eggenschwiler developed the Java version, then it became an open-source project. The original HotDraw was one of the first projects specifically designed for reuse and labeled as a framework. It was also one of the first systems documented in terms of design patterns [Joh92].

JHotDraw defines a basic skeleton for a GUI-based editor with tools in a tool palette, different views, user-defined graphical figures, and support for saving, loading, and printing drawings. The framework has been used to create many different editors from CASE tools to a Pert diagram editor [JHob].

We chose for comparison version 4.0 and 5.0. The purpose of 5.0 release was to use features (new at that time) available in JDK 1.1 like event model, access to resources, serialization, scrolling, printing, etc. It also consolidated the packaging structure and improved support for connectivity of diagrams.

2.2.2 Collecting the Data

The case study components are medium-size to large (e-Mortgage is 50 KLOC, Eclipse is roughly 2 million LOC). It is hard to discover the changes in a large system and many authors suggest that tools should be used to detect changes. For instance, Demeyer et al. [DDN00] used metrics tools to discover refactorings. However, most API changes follow a long *deprecate-replace-remove* cycle to preserve backward compatibility. This means that on obsolete API can coexist with the new API for a long time.

Consider a change such as renaming class `Category` to class `Logger` in log4j. In order to maintain compatibility with old clients, class `Logger` (the new name) inherits from class `Category`. The constructor of `Category` became protected so that users can not create categories directly but invoke in-

stead the creational method `getInstance()`. This method returns instances of the new class `Logger`. Any method in `Category` that returned an object of type `Category` became deprecated. Clients should replace all the references to `Category` with references to `Logger`. The two classes still coexist, but `Category` will be deleted eventually. Such a three-step change would have been misinterpreted by a tool, but a human expert can easily spot this as a renaming. At the time of this study, none of the tools available could detect refactoring because of the "noise" introduced by various techniques striving to maintain backwards compatibility. Later we developed RefactoringCrawler, the first tool that could operate under such conditions.

Second, we wanted a qualitative and quantitative study of all API changes that are not backwards compatible. At that time, tool support for detecting and classifying structural evolution was very limited: only a few types of refactorings (mostly merging and splitting) were attempted to be detected. Therefore, a comprehensive qualitative analysis of the breaking changes requires instead manual analysis.

Even for the larger components, manual analysis was feasible because we started from the change logs that describe the API changes for each release. These logs were carefully compiled by the component developers. For Eclipse we used its help system[1], the documents called "Incompatibilities between Eclipse 2.1 and 3.0" and "Adopting 3.0 mechanisms and API". For Struts we studied the "Release Notes" for version 1.2.4[2]. For Log4J we studied "Preparing for log4j version 1.3"[3]. For JHotDraw we studied the "Release Notes" for version 5.0, packaged along with the documentation of the framework.

Sometimes the release documents would be vague, reading for example "method M in class X is deprecated". Because of the deprecate-replace-remove cycle, many types of changes are masked by the deprecation mechanism. In those cases we read and compared the two versions of the source code in order to discover the intent behind the deprecation. When a method is deprecated it merely delegates to its replacement method. By reading the code we learned whether the new method is just a renaming of the deprecated method, whether the intent was to move the method to another class or whether the deprecated method was replaced by a semantically equivalent method that offers better performance.

For the e-Mortgage framework, for two days we interviewed the framework and application developers and then studied the source code. We classified all the breaking API changes from the case studies into

[1] Section: Eclipse 3.0 Plugin Migration Guide
[2] http://struts.apache.org/userGuide/release-notes-1.2.4.html
[3] http://www.qos.ch/logging/preparingFor13.jsp

structural and behavioral changes (qualitative analysis), then we recorded how many times each type of change occurred (quantitative analysis).

We double-checked the quantitative aspect of our analysis by using a tool, Van [GDL04] (a history analysis tool), and heuristics (like in [DDN00]). For each type of refactoring, we wrote queries in Van that return those structures suspected of that specific refactoring. For example, to detect changes in method parameters' types, we searched for methods that have the same name in both versions of a class, have the same number of arguments, have the same return type but have different signature. After analyzing and eliminating the false positives, the remaining candidates were found among those that were already documented in the change logs. Van found a few other places suspected of refactoring, but their number is less than 4% of those detected by starting from the change logs. Also Van failed to detect several refactoring candidates. This happened because of the noise introduced by the deprecate-replace-remove cycle described above. We could only cross reference our results for Struts and log4j. The tool did not scale up for Eclipse and we do not own the source code of the proprietary e-Mortgage framework. JHotDraw case study was added at a later date and since Van did not produce new data for the previous case studies, we decided not to run it on JHotDraw.

2.3 How APIs Change

In this section we classify the API changes in respect to how they affect backwards compatibility. The first subsection talks about APIs and what does it mean for an API change to break compatibility with applications. In Subsection NON-BREAKING API CHANGES, we describe some techniques used to change APIs without breaking applications. Subsection BREAKING API CHANGES presents the empirical data gathered from the API-breaking changes we noticed in the five case studies.

2.3.1 API Changes and Compatibility

An API is the interface that a component provides to application developers and its description is part of the component's documentation. The term has been extended to mean any component that is supposed to be reused by clients and thus is expected to be stable.

APIs make use of the visibility rules of the language in which the component was implemented. For instance in Java or C++ only members that are declared public or protected can be part of the API. However,

not all classes or class members that are public are intended to be reused by applications.

Usually there are no language features that distinguish between public entities that are intended to be part of the API and public entities that are not. Naming conventions can be used to identify those components that are "published" (to be reused) from those components that are "public" but are not intended to be reused [DDN03]. For instance, Eclipse places a public class that is not API in a package with "internal" as a prefix. Such a class is fair game to change without notice.

Over time, changes are made to APIs or APIs' behavior. Depending on whether or not they are backwards compatible, API changes can be classified as NON-BREAKING API CHANGES or BREAKING API CHANGES.

A **breaking change** is not backwards compatible. It would cause an application built with an older version of the component to fail under a newer version. If the problem is immediately visible, the application fails to compile or link. Or the application might compile fine, but behave differently at runtime. By behavior we mean functional behavior, e.g. the set of observable outputs for a given set of inputs. If the only observable difference is that an application is slightly faster or slower or has a different memory footprint, we do not consider it a breaking change[4].

A **non-breaking change** is backwards compatible. Such a change can be an enhancement like addition of new modules to extend the functionality of the component. Or it can be a performance optimization or an error removal.

A seemingly non-breaking change such as fixing a bug in the component might be a breaking change. If the application developers worked around the bug, then when the bug is removed from the component, the application might behave differently.

Although there are a number of techniques used to facilitate component changes without breaking the clients, breaking API changes happen all the time. It is these changes that cause problems for application developers. Our ultimate goal is to provide upgrading tools that can incorporate breaking changes.

2.3.2 Non-breaking API Changes

This subsection discusses changes to the API that preserve the backward compatibility of a component. We also present techniques that component developers use to maintain compatibility with older applications.

[4]We go with a light definition of failure but in domains like embedded systems that have hard real-time constraints, our notion of reliability needs to be extended

Usually new features of a component are packaged in separate modules and do not affect the existing applications. Developers still have to learn the new APIs in order to use the extensions.

Other times, component designers successfully employ established software engineering techniques such as information hiding, encapsulation and abstraction to introduce changes seamlessly. Such "under the hood" changes often include performance optimizations, security improvements, bug fixes and other implementation details. For instance, the financial institution that we visited made a big architectural change to their enterprise framework. For performance reasons they replaced EJB entity beans with Data Access Objects. They employed information hiding to isolate the changes from the web services that use the framework. All the services are calling a Session Bean Facade that offers the same interface to the outside world. For the application developers these changes happened seamlessly.

In the five case studies we noticed a number of other techniques used to facilitate API changes without breaking the client applications:

1. DEPRECATION is used to gracefully degrade old APIs and to warn application developers that certain APIs won't be supported in future versions. Developers can plan ahead for migration and avoid a last minute rush. The parts of the API that are obsolete are marked with special annotations (e.g. `@deprecated` tag in Javadoc). The deprecated API should also provide a description of how applications would migrate to the new API.

2. DELEGATION. Sometimes it is possible to make the old API implementation delegate to the new implementation so that clients make use of the enhancements. Nevertheless they should plan to migrate to the new API. For instance, this technique is often used when renaming a method in Struts. In class `ActionServlet`, method `destroyApplications` changed its name to `destroyModules`. In order to preserve backward compatibility, the class supports both methods:

```
1  /**
2   * @deprecated
3   * replaced by destroyModules()
4   */
5
6  protected void destroyApplications() {
7          destroyModules();
```

```
}
```

3. NAMING CONVENTIONS. Eclipse developers use this style when they extend an interface. For instance the initial version of interface IMarkerResolution was:

```
public interface IMarkerResolution {
    public String getLabel();
    public void run(IMarker marker);
}
```

   The new interface adds more methods without breaking existing implementations of `IMarkerResolution`:

```
public interface IMarkerResolution2 extends IMarkerResolution {
    public String getDescription();
    public Image getImage();
}
```

   This "numbered interface" naming convention suggests an extension to the original interface. The new interface can be used wherever the old one was expected. New clients can use the extended interface. However, old clients that want to use the new methods need to downcast.

4. RUNTIME SWITCH. Dealing with removed classes or removed methods from an interface is a bit trickier. Eclipse provides a Compatibility plugin that "adds back" at runtime the removed entities. For instance, Eclipse 2.1 API offered a method that is removed in version 3.0:

```
public interface IWorkbenchPage {
    IEditorPart openEditor(IFile file); // to be deleted
    ...
}
```

   Eclipse 3.0 checks whether a client application is compliant with version 2.1 or 3.0. When an older client makes a call to the removed method, this call is dispatched to an instantiator of the older version of the interface (which is added back and loaded up at runtime).

5. COM-STYLE INTERFACE QUERY. A component provides multiple versions of an interface. A client will first query for an interface that it knows how to interact with. If the component still

supports the older interface that client is requesting, the client and the component can communicate. In Eclipse, the initial document interface `IDocument` has acquired several extensions called `IDocumentExtension, IDocumentExtension2, IDocumentExtension3`. A client checks whether the document returned by `getDocument()` has an interface that it knows how to talk to.

2.3.3 Breaking API Changes

Breaking changes are extremely disturbing in the development life cycle of component-based applications. When application engineers are in the middle of development, upgrading a component could hurt costs and schedules. Unless there is a high return-on-investment, application developers will not want to migrate to the new version of the component [Lai99].

Table 2.2 lists the types of BREAKING API CHANGES that we observed in the components that we studied. The first column identifies the type of change. Those changes in *italic* font are refactorings. The remaining columns give the number of times each type of change occurred in the components. Columns Eclipse* (E*) and Struts* (S*) deal with "recommended" changes. Component designers marked these as changes that will be enforced in the next major release. Even though technically these are not breaking changes for the current release (they were insulated by the deprecation mechanism), we included them to show the trend of breaking changes that are coming in next versions. Based on how many times each type of change occurred, we sorted the rows so that most popular changes appear first.

2.4 A Catalog of Breaking API Changes

This section categorizes the changes in Table 2.2 according to how they affect the semantics of the component. The structural transformations are semantic-preserving changes (refactorings) while the behavioral changes are semantic-modifying.

An API class can have two types of clients: *instantiators* and *extenders*. As for API methods, due to extensive usage of callbacks (hook methods) in frameworks, there are two types of clients of API methods: *callers* and *implementors*. Some changes affect both types of clients while other changes affect only one type of clients.

JHotDraw illustrates some of the subtle effects of API changes on client code. `AbstractFigure`

Type of change	E	E*	M	S	S*	L	JHD
Moved Method	16	13	-	11	28	9	-
Moved Field	-	45	-	18	2	5	-
Deleted Method	2	2	-	24	32	-	2
ChangedArgumentType	5	-	4	18	4	11	-
Changed Return Type	2	-	1	2	-	2	31
Replaced Method Call	1	20	-	8	4	-	1
Renamed Method	4	-	-	16	5	8	-
New Hook Method	4	2	2	7	-	-	5
Extra Argument	3	2	2	1	1	-	2
Deleted Class	-	-	-	9	-	-	1
Extracted Interface	-	-	-	-	-	-	7
Renamed Field	-	-	-	6	1	-	-
Renamed Class	-	1	-	2	-	2	2
Method Object	3	-	-	-	-	-	-
Pushed Down Method	3	-	-	-	-	-	-
Moved Class	-	2	-	-	-	-	1
Pulled Up Method	-	-	-	1	-	-	-
Renamed Package	-	-	-	-	-	-	1
Split Package	-	-	-	-	-	-	1
Split Class	-	-	-	-	-	-	1
New Method Contract	3	12	1	8	-	1	-
Implement New Interface	1	-	1	5	-	-	3
Changed Event Order	3	-	-	-	-	-	-
New Enum Constant	1	-	-	-	-	-	-

Table 2.2: Types of BREAKING API CHANGES and the number of these API changes in Eclipse (E), e-Mortgage (M), Struts (S), log4j (L), and JHotDraw (JHD). Eclipse* (E*) and Struts* (S*) denote recommended changes, that is changes that will become breaking changes in future releases. Those changes in italic font (upper half of the table) are refactorings.

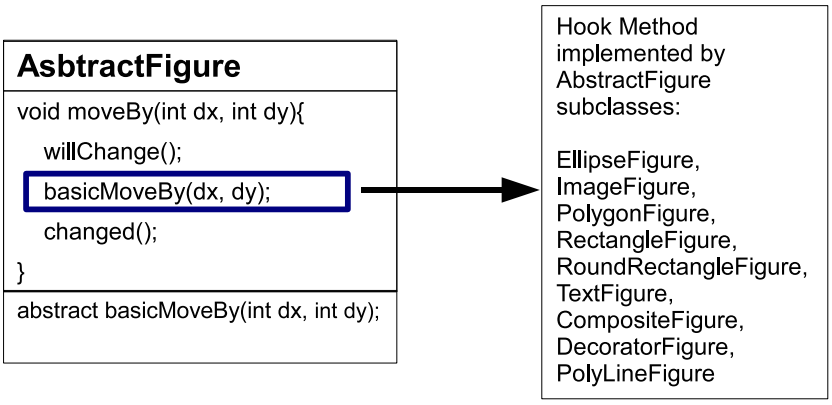

Figure 2.1: Using `basicMoveBy` Hook Method to introduce points of variability in several subclasses of `AbstractFigure` in JHotDraw framework.

provides default implementation for the figure classes. We encapsulate commonalities among different class figures, by using the Template Method design pattern [GHJV95]. Thus method `moveBy` (referred as template method) contains the fixed algorithm for moving a figure (see Fig 2.1): first it announces that a figure is about to change something that will affect its displaying box (by calling method `willChange`), then the figure is moved with the specified delta (by calling method `basicMoveBy`), and lastly it informs that a figure modified the area of its display box (by calling method `changed`). The sequence of these three steps is common for all the figures, what changes from figure to figure is encapsulated in the abstract method `basicMoveBy`. Therefore this method becomes the hinge point in the algorithm, or the *hook method*. Clients wishing to add new types of figures must implement this method. Rather than client code calling the framework, the framework will call the client-provided method when a user moves a figure (thus the callback nature of this method).

Any change that affects the signature or name of the hook method is potentially changing the behavior of client code. A change to the hook method that breaks the polymorphic overriding with a client method will result in the client method being skipped whenever the template method is invoked (assuming that the hook method is not abstract). Therefore, fewer client methods get called. The inverse can happen, too. A change to the hook method that makes it become the super method of a different client method will result in the client method being called (accidentally) whenever the template method is called. Steyaert et al. [SLMD96]

refer to the first case as *Inconsistent Method* and to the second as *Method Capture*. Since Method Capture and Inconsistent Method don't produce compile errors, one could fallaciously assume that the upgrading was safe. If the hook method is declared as an abstract method (as in our example), when the hook method changes so that there are no more client methods overriding it, the compiler signals an *Unimplemented Method* error in the client class.

Any of the Inconsistent Method, Method Capture or Unimplemented Method can occur, depending on the nature of change in the hook method and the existing methods in the client class. We give examples of all three scenarios using `BackgroundFigure`, a client class not part of the framework, but which subclasses `AbstractFigure` introduced earlier Figure 2.1. The client class `BackgroundFigure` contains, among other methods, the following methods:

```
1  public class BackgroundFigure extends AbstractFigure{
2      void basicMoveBy(int dx, int dy){
3          // overrides the hook method in the superclass
4          ...
5      }
6
7      void moveByDelta(int dx, int dy){
8          // a helper method
9      }
10 }
```

Recall that the framework designers do not have access to client class `BackgroundFigure` (actually they do not even know about its existence). If the framework designers move the `basicMoveBy` hook method to another class and update its call site in the `moveBy` template method, the `basicMoveBy` implementation provided by client `BackgroundFigure` no longer overrides the hook method. When the template method is called on an instance of `BackgroundFigure`, its `basicMoveBy` does not get called anymore, thus leading to Inconsistent Method. If instead framework designers rename the `basicMoveBy` hook method to `moveByDelta`, since there exists a method with this signature in the client class, this method will be (accidentally) called whenever the template method is invoked, therefore leading to Method Capture. Otherwise, if the framework designers rename the abstract hook method such that there is no method in `BackgroundFigure` that overrides it, the compiler would signal an Unimplemented Method

error when the older version of class `BackgroundFigure` is put together with the newer version of `AbstractFigure`.

2.4.1 Refactoring Transformations

To improve reusability and maintainability, components are often refactored. Refactorings affect only the structure of the component and are meant to preserve the functional behavior of the component. For example, consider what happens when an API method is renamed.

Component designers rename an instance method in the component. They find and update all the callers and implementors of the method to reflect the new name. For the component itself this change is safe and does not modify its behavior. However, it is not backwards-compatible: client applications that call the renamed method are broken. Thus a behavior-preserving change (refactoring) for the component might lead to a breaking change for the application.

Most times application code is not available to component developers when they make structural changes. The result is that applications might not even compile with the new version of the component. Once the application developer solves the compile errors, the application's behavior is the same (structural changes in the component do not introduce new behavior). In cases when the structural changes accidentally induced modified behavior (like in Method Capture or Inconsistent Method described above), the application developer will have to make semantical code changes to match the changes in the component.

Next we describe all types of API-breaking changes in the five case-studies. For each component refactoring we explain why it can break the older versions of the applications and what errors would an application developer encounter when trying to use the new version of the component along with the older version of the application. In later chapters (Chapters 3 - 6) we present our tools that can automatically incorporate the effects of the component refactorings into applications.

Moved Method. The most common way that instance methods moved in Eclipse is by becoming static methods in new host classes. The rationale was to factor out the non-IDE specific methods into utility classes to preserve the convenience of the old methods. Usually the moved method takes the old home class as an extra argument. This ensures that the moved method can access public members in the old home class.

In Struts, instance methods remain instance methods after they move to other classes. Old callers of the

method ask a factory method for an instance of the new home class and then call the moved method. Other ways that methods are moved are variations of the Move Method refactoring described by Fowler [FBB+99].

Implementors of the moved method would not compile if they make a call to the super() method. As for the callers of the moved method, they might compile or not depending on whether the method is used polymorphically. If the moved method is declared in a parent class and is overriden in the client class and the method is invoked by sending a message to an object having the type of the client class, the behavior is preserved. However, good OO principles recommend that instance or local variables are declared having the most general supertype. In this case, in a statically-typed language, a compile error will warn that there is no such a method in the parent class. Moving a hook method out of the parent class results in Inconsistent Method (described above) since the client method overriding the hook is no longer called.

Moved Field. Encapsulation requires that the variables that characterize the state of an object are not exposed. However, sometimes fields are publicly exposed either because of convenience or because they represent constants. When fields are placeholders for global constants usually they are declared as static fields. In Eclipse, Struts and log4j, only fields that were constants moved to another home class. Moving a public field results in compile errors for the clients who access it.

Deleted Method. Typically this happens after a method is renamed or moved to another class. For compatibility reasons, component producers support both the old and new method for a while. After all the references to old method were replaced, the method is deleted since it is obsolete API. This usually results in compile errors for the clients calling the method. Deleting a hook method results in the Inconsistent Method effect since the client method overriding the hook is no longer called.

Changed Argument Type. We observed several kinds of argument type changes.

1. The type of a method argument is replaced with its supertype to make the method more general. This change may or may not break an existing application depending on whether the application calls any methods that are not visible through the supertype's interface.

2. The type of method argument is replaced by another type while the relationship between the two is aggregation. This is often the case when replacing a primitive type with an object type (e.g. in

Java replace *int* with *Integer*). Another special case is replacing a type with a collection that contains several elements of the previous type. In order to regard these changes as automated refactorings, one needs to know how to access the member from the wrapper and how to get the proper wrapper for a member. In the e-Mortgage framework the method `process(String message)` changed to `process(Envelope e)` with `Envelope` encapsulating the `message` string. Callers of `process` have to pass an Envelope instead which is obtained from a factory method. The implementors of `process()` should augment their implementation to match the new type. They will first obtain the String message out of the Envelope. This change results in compile errors for the clients calling the method. If the arguments of a hook method are changed, this can result in any of Method Capture, Inconsistent Method or Unimplemented Method.

Changed Return Type. This change is very similar to CHANGED ARGUMENT TYPE. We observed one interesting change in Eclipse. The return type of `IJavaBreakpointListener.breakpointHit()` was changed from boolean to integer to allow listeners to vote "don't care" in addition to "suspend" and "don't suspend". A refactoring tool can only swap primitive types if there is a translation map between the values of the two different types. For the callers that assign the result of the method call or pass it to other method calls, this type of change can produce compile errors depending on what is the relationship between the new returned type and the expected type. For implementors of the changed method, due to the fact that statically types languages like Java do not allow two methods in the same class to differ only by the return type, accidental Method Capture or Inconsistent method cannot occur. The compiler signals an error that the overriden method and the super class method differ in the return type.

Replaced Method Call. The clients of a method should call another method that is semantically equivalent and is offered in the same class. When there are no more callers to the original method, it is usually deleted. In Struts for example, clients of `FieldChecks.validateRange(..)` should call instead `FieldChecks.validateIntRange(..)`. If the replaced method was a hook method, this change can result in any of the Method Capture, Inconsistent Method or Unimplemented Method.

Renamed Method, Renamed Class, Renamed Field, and Renamed Package are used to give intention revealing, self-explanatory names to methods, classes, fields, and packages. These refactorings are well

described in refactoring catalogs (see [FBB+99]). Renaming a method usually results in compile errors for the callers of the method. Renaming a hook method can result in any of the Method Capture, Inconsistent Method or Unimplemented Method. Class renamings result in compile errors for both the instantiators and implementors of the class. Field renamings result in compile errors for the clients that access them. Package renamings result in compile errors for clients that use entities from the renamed package.

New Hook Method. Component producers factor out a method to provide "hot spots" that need be specialized by subclasses (see Template Method in [GHJV95]). They add a new hook method in the super class (usually as an abstract method) that all non-abstract subclasses must override. We illustrate this with an example from Struts. Method `validate()` in class `ValidatorForm` calls the newly introduced method `getValidationKey()`:

```
1  public ActionErrors validate(ActionMapping mapping,HttpServletRequest request)
2  {
3      ...
4      String validationKey =
5          getValidationKey(mapping, request);
6      ...
7  }
8
9  String getValidationKey(ActionMapping mapping, HttpServletRequest request){
10         return mapping.getAttribute();
11 }
```

Subclasses override `getValidationKey()` to provide the desired behavior. It might happen that an existing subclass already has a method with the same signature as the newly introduced hook method. In this case the method provided by the inheritor gets captured by the parent class even though the inheritor did not intend this, therefore Method Capture occurs. Using a refactoring tool to perform this change would warn when method capture happens.

Extra Argument. Often two methods signatures are very similar, they only differ by an argument. The two methods do similar things but one method can do extra things by making use of the extra argument. When eliminating duplicated code, usually the method with fewer arguments will be replaced by the one

with more arguments. For the call sites of the displaced method, this change appears as if the method gained one more argument. The callers of the old method with fewer arguments will have to call the new method and pass a default value for the extra parameter.

Developers of the e-Mortgage framework decided that database connections should be reused from a connection pool rather then being created every time a database operation was required. In order to persist an object one would call the following method in the framework:

```
boolean persist(BusinessObject)
```

Inside `persist` method a database connection would be created. The later version of this method looks like:

```
boolean persist(BusinessObject, DBConnection)
```

When a web service calls this method it needs to pass along an existing database connection (in case that it owns one). When the null object is passed, the `persist` method will create a connection on the fly.

This change usually produces compile errors for the clients calling the method. When a hook method gains more arguments, any of the Method Capture, Inconsistent Method or Unimplemented Method can occur.

Deleted Class. Component producers delete a class when it is no longer supported or maintained due to lack of resources or because the implementation is too error-prone. In Struts several classes acted like containers for particular objects. The container's name would suggest that it contains objects of a certain kind(e.g. ActionMappings holds a collection of ActionMapping objects). In a later version the containers are superseded by general-purpose collection classes and then deleted. Class deletion results in compile errors for both the instantiators and implementors of the class.

Extracted Interface. Component developers extract the signatures of the public methods offered by a class into an interface (in Java, interfaces are first-class entities). Clients of the old class should call its methods through the interface. This change is meant to make component code easier to extend: new classes implementing the interface can be passed as a concrete implementation of the interface, without having to change client code. In JHotDraw 5.0 all the key abstractions are extracted from their previous

classes as interfaces. The interface takes the name of the class from which it was extracted. The old class implements the interface in one of two ways: as abstract class or as standard class. Abstract classes (like `AbstractFigure`) provide default implementation but still need to be subclassed. Standard classes (like `StandardDrawing`) can be used as are. The new prefix of the class ("abstract" or "standard") makes it easy to distinguish between them. This change does not affect clients that were only invoking methods of the class. However, inheritors of the old classes should be changed to inherit from the corresponding abstract or standard classes, otherwise Method Unimplemented errors are thrown by the compiler.

Method Object. This is a variation on Method Object described by Kent Beck [Bec97] and we illustrate it with an example from Eclipse. In class `AbstractDocumentProvider`, the modifier of `saveDocument()` method changed to *final* so that subclasses cannot override it anymore. A new method called `doSaveDocument()` was introduced and all the code from `saveDocument()` moved to the new method. A `DocumentProviderOperation` object offers an `execute()` method that delegates to `doSaveDocument()`. The new implementation of `saveDocument()` creates an instance of the `DocumentProviderOperation` and then calls its `execute()` method. All previous implementors of `saveDocument()` must override `doSaveDocument()` instead.

Pushed Down Method. A service is no longer offered by the superclass but only by subclasses. Thus we say that the corresponding method was pushed down in the class hierarchy. This change results in compile errors for the old method clients that call the moved method through the interface of its original class. However, a much subtler behavioral change can occur without being spotted by the compiler. Imagine the scenario in Fig. 2.2. On the component side, the superclass `A` defines method `m`. This method is overridden in the subclass `B`. Class `C` is yet another component class that inherits from `B`. On the application side, class `D` inherits from `B` and therefore any call to `super.m` gets dispatched to the implementation provided by `B`. In the next version of the component, without being aware of the presence of client class `D`, component providers push down the implementation of `B.m` from `B` to `C`. The client class `D` compiles fine, but when it calls `super.m`, the call gets dispatched to the implementation provided by `A`, thus leaving to a different behavior than when using the previous release.

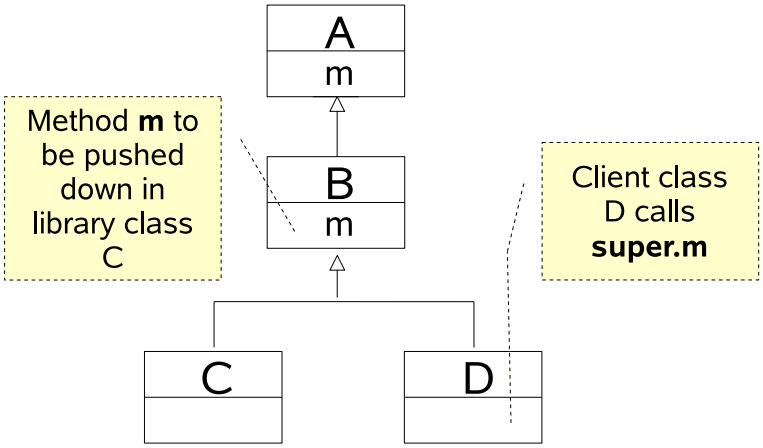

Figure 2.2: When Pushed Down Method can introduce a behavioral change.

Moved Class. A class is moved to a different package in order to increase the cohesiveness of that package. This change breaks both the instantiators and implementors of the moved class.

Pulled Up Method. A method is moved in the parent class so that everyone can take advantage of the superclass logic. This change will not produce any compile errors, but can lead to an incorrect behavior. Consider the scenario in Fig. 2.3. Before doing the upgrade, when the client class `D` calls `super.m`, this call gets dispatched to the implementation provided by superclass `A`. In the next component version, `C.m` gets pulled up from `C` to `B`. When client `D` calls `super.m`, this is dispatched to the implementation provided by class `B`.

Split Package. In order to enhance the cohesiveness of classes in a package, component designers split a package into smaller, more cohesive packages. In JHotDraw, a `figures` package was removed from the `standard` package. This package provides a kit of standard figures and their related handle and tool classes. This change results in compile errors for both instantiators and implementors of those classes belonging to the new package.

Split Class. Each class should have only a few responsibilities. When a class acquires too many responsibilities, the initial class cohesiveness will be fractioned into clusters of methods and fields that interact with

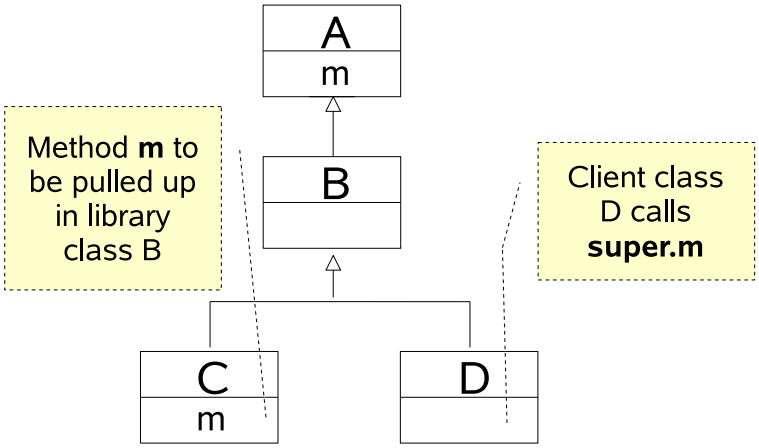

Figure 2.3: When Pulled Up Method can introduce a behavioral change.

each other closely. Component designers choose to "purify" the cluttered class by moving responsibility to other classes, usually though object composition or inheritance. In JHotDraw `AbstractFigure` was split. It no longer keeps track of the attributes of a figure. The bookkeeping of attributes happens now in a subclass called `AttributeFigure`. Clients who were calling or implementing attributes-related API methods no longer compile.

2.4.2 Behavioral Modifications

We saw that structural transformations preserve the behavior of the component but might cause applications to fail to compile. In contrast, behavioral modifications in the component might cause the application to compile fine with the new version. However, the application will not behave the same since the new version makes different assumptions.

New Method Contract. A contract is an agreement between the method provider and its clients [Mey05]. The precondition is what the method assumes to be true before execution. A postcondition is what a method guarantees to be true after the method body has executed successfully (presuming that the precondition holds).

Des Rivieres [Riv] shows the effect of strengthening or weakening preconditions and postconditions

What changes	How	Callers	Implementors
Precondition	weaken	compatible	**broken**
Precondition	strengthen	**broken**	compatible
Postcondition	weaken	**broken**	compatible
Postcondition	strengthen	compatible	**broken**

Table 2.3: Effects of Changing Method Contract on Callers and Implementors.

on clients of a method as seen in Table 2.3. The first column identifies what part of the contract changes. The second column gives the direction of change: strengthening or weakening the contract. The next two columns show whether the change is backwards compatible or it breaks existing method clients.

Consider the following method offered by the Collection interface:

```
1 /** @param coll a non-null Collection */
2 public boolean addAll(Collection coll);
```

Component designers consider weakening the precondition so that it is acceptable to pass a null object. The callers of this method are not affected. However, an implementor like the one below will throw a `NullPointerException` when it sends `size()` message to a null object:

```
1 public boolean addAll(Collection coll){
2     //an implementation
3     int size= coll.size();
4     ....
5 }
```

If the precondition was strengthened (e.g., the collection passed as an argument should contain at least one element), some existing callers of the method might not fulfill the requirements thus causing some faulty behavior. The existing implementors are not affected since they assumed less than what is offered now.

Implement New Interface. Developers of the component replace the interface implemented by a class with a different interface (with different contracts). Or they add a new interface to the ones a class already implements. In Struts, the latest version of class `LabelValueBean` implements a new interface, namely `Comparable`. The class now overrides methods `compareTo(Object)`, `equals(Object)` and `hashCode()`. Older applications that compared instances of this class using the default implementation inherited from superclass `Object` might have different behavior now that the class provides its own

equality checks.

Changed Events Order. Similar to orchestra conductors, frameworks control the code contributed by applications. Usually the applications just respond when the conductor gives them the signal to participate. When the application make assumptions about the order in which the events are generated it is vulnerable to any change in the sequence of events. For instance in Eclipse 3.0, selection of items in tables and trees generates the event sequence MouseDown-Selection-MouseUp. In version 2.1 the event order was different under some platforms with Selection event being generated first, i.e. the sequence Selection-MouseDown-MouseUp.

New Enumeration Constant. This change affects clients that rely on the set of all possible fields in an enumeration. In Eclipse 2.1, IStatus is an enumeration with four constants: OK, INFO, WARNING and ERROR. Some clients used a switch case statement to check all the values of an enumeration. They treated the ERROR case in the *default* branch of the switch statement. Eclipse 3.0 adds a new constant, namely CANCEL. When CANCEL is passed around, these old clients process the new constant in their *default* branch thus accidentally overlapping with the ERROR case.

Miscellaneous. Besides API changes there are other types of changes that may cause component-based applications to malfunction. Some of these changes might be: deployment changes, classloader order changed, changes to build scripts and other configuration files, data format and interpretation. We noticed changes in the XML configuration and metadata files in all four studied frameworks. However, because these changes do not affect the APIs per say, they are beyond the scope of this study.

2.4.3 Summary of Findings

Table 2.4 is a summary of Table 2.2. The first column lists the components we studied. As we did in Table 2.2, Eclipse* and Struts* denote recommended changes, that is changes that will become breaking changes in the next release. The second column gives the total number of breaking API changes (both structural and behavioral). The last column shows how many of the breaking API changes are refactorings.

Our findings suggest that most API breaking changes are small structural changes. This makes sense because large scale changes lead to clients abandoning the component. For a component to stay alive, it

Component	# Breaking Changes	% Refactorings
Eclipse	51	84%
Eclipse*	99	87%
e-Mortgage	11	81%
Struts	136	90%
Struts*	77	100%
Log4J	38	97%
JHotDraw	58	94%

Table 2.4: Ratio of refactorings to all breaking API changes. Eclipse* and Struts* denote recommended changes (e.g., API changes that will be enforced in future versions although for the current version client applications are insulated through the deprecation mechanism).

	Struts	Log4J
Refactorings	123	37
All Other API Changes	325	920
Ratio of Refactorings to other API changes	27.4%	3.8%
Impact of Refactorings upon backwards compatibility	90%	97%

Table 2.5: For Struts and JHotDraw, refactorings represent a relatively small percentage (see third row) of all API changes (including addition of new APIs). However, refactorings make up the majority of API changes that are not backwards compatible (see fourth row).

should change through a series of rather small steps, mostly refactorings.

For Struts and log4j we analyzed what percentage of all API changes (including addition of new APIs) are represented by refactorings (see Table 2.5).

We used Van [GDL04] to learn the number of addition and deletion of API classes and methods. The second row sums the API methods that were added or deleted from classes that exist in both versions, the number of API classes that were added or deleted in between the two versions, and the number of breaking API changes that are not refactorings. Row ‘Percentage of Refactorings’ depicts how many of all API changes (including non-breaking changes like addition of new APIs) are refactorings. Row ‘Impact of Refactorings’ depicts how many of all changes that break existing customers are refactorings. Table 2.5 shows that even though refactorings are a small percentage of all API changes (including addition of new APIs), they have a large impact upon backwards compatibility. Therefore, upgrading tools should focus on these types of changes.

2.5 Summary

API changes have an impact on applications. One might argue that component developers should maintain old versions of the component so that applications built on those versions continue to run. However, this results in version proliferation and high maintenance costs for the producer. In practice, it is application developers who adapt to the changes in the component.

We looked at one proprietary framework, three Open-Source frameworks and one Open-Source library and studied what changed between two major releases. Then we analyzed those changes in detail and found out that in the five case studies, respectively 84%, 81%, 90%, 97%, and 94% of the API-breaking changes are structural, behavior-preserving transformations (refactorings).

There are several implications of our findings. First, they confirm that refactoring plays an important role in the evolution of components. Second, they offer a ranking of refactorings based on how often they were used in the five systems. Refactoring vendors should prioritize to support the most frequently used refactorings. Third, they suggest that component producers should document the changes in each product release in terms of refactorings. Because refactorings carry rich semantics (besides the syntax of changes) they can serve as explicit documentation for both manual and automated upgrades. Fourth, upgrading tools should focus on support to integrate into applications those refactorings performed in the component.

Chapter 3

Refactoring-aware Upgrading Tools

Chapter 2 showed that most of the API-breaking changes in five widely-used components were caused by refactorings. If all the breaking changes were refactorings, it would clearly make sense to use a refactoring tool to incorporate these changes in the applications. However, if tools could handle even 80% of the changes, they would reduce the burden of manual upgrades. Application developers would have to carry out only a small fraction (less than 20%) of the changes. These are changes that require human expertise.

This chapter describes common terminology used in the remaining chapters. We are assuming that the reader is familiar with the object-oriented terminology (e.g., method, method signature, method overriding, class, class inheritance). First, the chapter presents refactorings as program transformations. Then it gives a gentle introduction to refactoring engines. It also describes the requirements for an upgrading tool based on refactorings. A practical solution should not alter the development process of components, and should be safe and automated for application developers. We conclude the chapter by presenting a technique, record-and-replay of refactorings, and an extension to the refactoring engine to use this technique.

3.1 Background on Refactorings as Program Transformations

Any change to programs can be regarded as a function from programs to programs, more precisely, a program transformation $T : Program \rightarrow Program$.

Definition 1 (Program operations) *Program transformations are composed of primitive program transformations, which we call program operations, denoted with τ. There are two types of program operations: those that change the semantics of the program, and those that do not. We refer to refactorings as a subset of operations that preserve semantics of a program, while edits change semantics (see Figure 3.1).*

Operations usually have preconditions: adding a method to a class requires that the class exists and does not already define another method with the same name and signature, while changing the name of a method

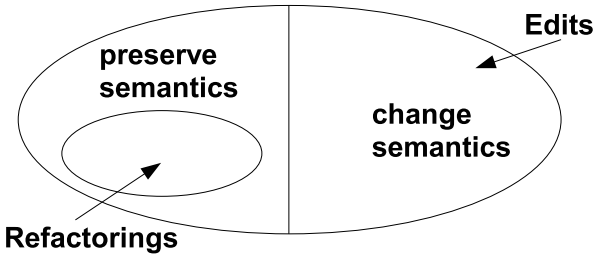

Figure 3.1: Refactorings are operations that preserve semantics of a program, while edits change semantics.

requires that the new name is not in use. Adding a new call site is an edit that requires the method to be defined.

Applying an operation τ inappropriately to a program P results in an invalid program, represented by $\perp$. For any operation τ, $\tau(\perp) = \perp$.

$$\tau(P) = \begin{cases} P' & \text{if preconditions of } \tau \text{ hold} \\ \perp & \text{if preconditions of } \tau \text{ do not hold} \end{cases}$$

A program transformation is a sequence of operations. The composition of two operations is denoted by ";":

$\tau_1; \tau_2(P) = \tau_2(\tau_1(P))$

Note that the composition operator ";" also models the precedence: $\tau_1; \tau_2$ means first apply τ_1 and then apply τ_2 on the result.

Definition 2 (Operations commutativity) *Two operations* commute on a program P *iff applying them in either order produces the same valid program* P''*:*

$\tau_2; \tau_1(P) = \tau_1; \tau_2(P) = P'' \wedge P'' \neq \perp$

Definition 3 (Operations conflict) *Two operations* conflict *with each other on a program* P *iff applying them in either order produces an invalid program:* $\tau_2; \tau_1(P) = \perp \wedge \tau_1; \tau_2(P) = \perp$

For example, adding two methods with the same name and signature in the same class results in an invalid program.

When two operations do not commute on a program P, we say that there is an *ordering dependence* between them. We denote this ordering dependence with the $\prec_P$ symbol.

Definition 4 (Operations dependence) τ_2 depends *on* τ_1 *on a program* P *(*$\tau_1 \prec_P \tau_2$*) iff* τ_2 *and* τ_1 *do not commute:*

$\tau_1 \prec_P \tau_2$ *iff* $\tau_1;\tau_2(P) \neq \perp \wedge (\tau_1;\tau_2(P) \neq \tau_2;\tau_1(P))$

For example, consider a program that initially defines three overriding methods `m` in the same inheritance class hierarchy. A developer adds a fourth method `m` in a class belonging to the hierarchy, intending to override the other `m` methods. Another developer intends to rename all overriding methods `m` to `n`. $\tau_1 = AddMethod(m, C)$ and $\tau_2 = RenameMethod(m \rightarrow n)$. If RenameMethod is executed before adding the new `m`, the final program contains three methods `n` and one method `m`. However, this contradicts the intent of having all four methods overriding each other. On the other hand, if adding method `m` is applied before the renaming, the final program contains four methods `n`, all in an overriding relationship. Although both orders produce valid programs, it is the latter that preserves the intent and semantics of the developers, thus $\tau_1 \prec_P \tau_2$.

The $\prec_P$ dependence is *strict* partial order, that is, it is irreflexive, asymmetric, and transitive.

Let us now revisit the definition of a refactoring operation from Opdyke [Opd92].

Definition 5 (Refactoring with Pre-conditions) *A* refactoring operation *is a behavior-preserving program transformation with a precondition that the program must satisfy before the transformation can be applied. More formally, a refactoring is a pair* $R = (pre, T)$*, where* pre *is the precondition that the program must satisfy, and* T *is a program transformation.*

Roberts [Rob99] uses first-order predicate logic (FOPL) to specify preconditions. We follow his notation. Along with the existential quantifiers, we also use several predicates that operate on programs. Since most of these predicates are trivial to express, we do not expand the definition of a predicate unless it is not clear from its name what it denotes. In addition, since preconditions, predicates and quantifiers are evaluated on concrete programs, we denote as a subscript $_P$ the program on which the evaluation takes place.

For example, the precondition for a simple refactoring such as deleting a class `C` from a program `P` is that class `C` exists and is not referenced:

$pre(DeleteClass_P(C)) = C \in P \wedge \neg isReferenced_P(C)$

If these preconditions are met for a particular program `P`, then DeleteClass refactoring does not change the semantics of `P`. However, if the preconditions are not met, applying the transformation T would result in

an invalid program.

Although a refactoring might preserve the behavior of a single component, it might not preserve the behavior in the context of the component plus application.

Consider a larger program, `P'`, that contains a component `Comp` and an application `App`. We use the notation of Batory [Bat07] to represent:

`P'` = `Comp` + `App`

where the $+$ operator represents the set union[1] of all classes in programs `Comp` and `App`, assuming that there are no name conflicts among the classes in `Comp` and `App`.

When computing predicates for this larger program, `P'`, we need to create the disjunction of the predicates for the programs contained in `P'`. For example, the predicate $isReferenced$ becomes:

$$isReferenced_{Comp+App}(C) = isReferenced_{Comp}(C) \vee isReferenced_{App}(C)$$

Checking the preconditions of the DeleteClass refactoring for a larger program that contains both component `Comp` and application `App` and applying DeMorgan's law results in:

$$\begin{aligned} pre(DeleteClass_{Comp+App}(C)) &= C \in (Comp + App) \wedge \neg isReferenced_{Comp+App}(C) \\ &= (C \in Comp \vee C \in App) \wedge \neg(isReferenced_{Comp}(C) \vee isReferenced_{App}(C)) \\ &= (C \in Comp \vee C \in App) \wedge (\neg isReferenced_{Comp}(C) \wedge \neg isReferenced_{App}(C)) \end{aligned}$$

This precondition is not satisfied in an application that has a reference to class `C`. Had the component designers known about such an application, they would not have deleted class `C`. In the absence of such information, the deletion of class `C` is a refactoring from the point of view of the component (i.e., its preconditions hold for program `Comp`), though it is not a refactoring from the point of view of the application (i.e., its preconditions do not hold for program `Comp + App`).

Another scenario is when a refactoring's preconditions are met for both the component and the application, but the program transformation carried by the refactoring is applied on the incomplete code base of component without application. Consider a refactoring that renames an API method in the component, m_1 to m_2. Because of lack of access to application source code, the transformation T_{Rename} carried by this refactoring changes only the source code of the component, `Comp`, into `Comp'`. When putting together the

[1] conceptually, a program is the set of all class declarations

refactored version of the component, `Comp'`, and the older version of the application, `App`, this results in an invalid program if the application has at least one call site to method m_1.

Compilation errors are easy to catch. However, due to heavy use of polymorphism and method overriding in OO programs, the application might compile fine, but behave differently at runtime. As we saw in Section 2.4, a refactoring performed on an incomplete code base might introduce accidental method overriding or method capture. Such a change inadvertently affects the behavior of the application. The toolset that we develop ensures that the upgrading does not produce compile errors; it also protects against inadvertent changes of behavior. Because refactorings are currently the only transformations with well defined semantics that can be easily checked by tools, we can only offer such guarantees with respect to refactoring operations.

3.2 Background on Refactoring Engines

A refactoring engine is a tool that automates the application of refactorings. A developer chooses a refactoring and the engine automatically checks the preconditions. If they are met, the engine performs the transformation, otherwise it warns the user that the transformation might produce an invalid program or a program whose semantics are changed. Refactoring engines are a central part of modern IDEs and programmers rely on them to change large programs.

We illustrate a few concepts using the popular Java refactoring engine in Eclipse [Ecla]. The same concepts appear (with minor differences) in NetBeans [Net], another popular open-source IDE for Java released by SUN, and JavaRefactor [Jav], the first open-source refactoring engine for Java (we implemented and released this engine in 2001). The similarity between all these engines is due to the desire to imitate a very successful Smalltalk predecessor, the RefactoringBrowser, which was the first refactoring engine, and whose theory was grounded in Opdyke's work [Opd92].

A developer chooses a program element to refactor and a specific refactoring; from there on the refactoring engine does most of the remaining work. Figure 3.2 shows the interface for renaming a method using the refactoring engine in Eclipse [Ecla]. The user has the option to preview the program transformations that the refactoring engine would apply (see Figure 3.3).

The life cycle of a refactoring in Eclipse is as follows:

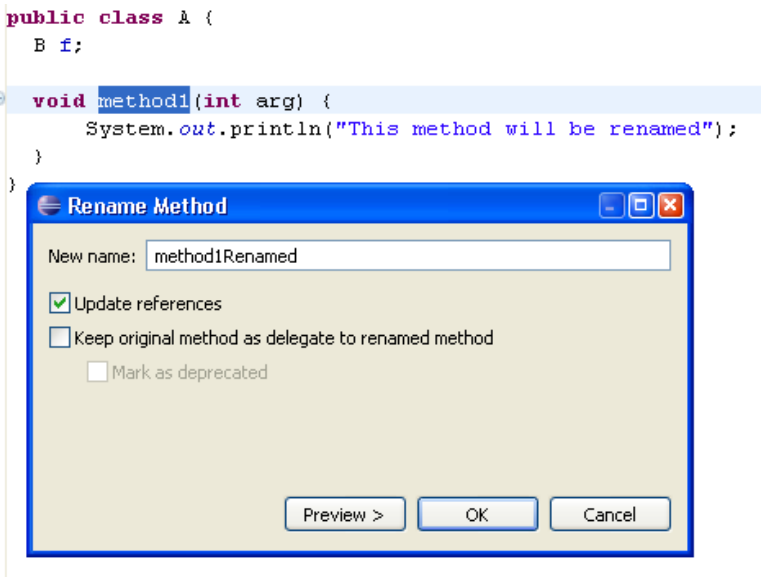

Figure 3.2: Renaming a method using the refactoring engine in Eclipse.

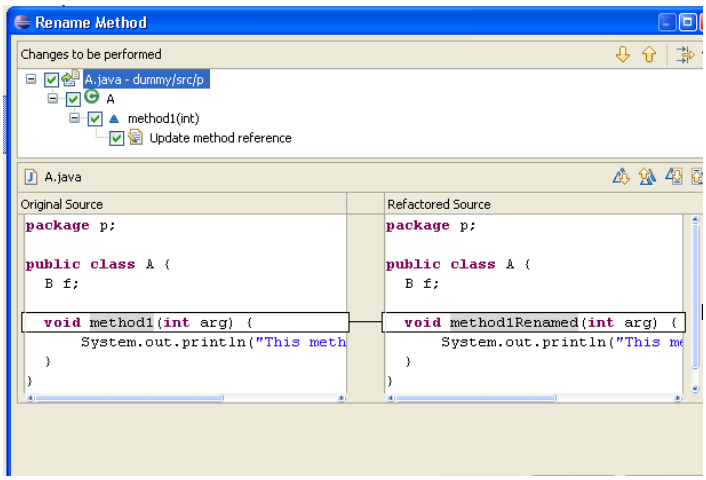

Figure 3.3: Previewing the changes before applying a refactoring in Eclipse.

1. The engine creates the refactoring object. A refactoring object is specific to each type of refactoring. A refactoring object is an example of the Command [GHJV95] design pattern, thus the engine delegates the precondition checks and the code transformation to the refactoring object.

2. The engine initializes the refactoring object with the program element to be refactored.

3. The engine calls `checkInitialConditions` on the refactoring object. This method efficiently checks for simple things that might prevent the refactoring from succeeding later (e.g., the source code is unavailable or is read-only protected). Assuming that this step proceeds fine, the refactoring advances to the next step.

4. The engine asks the user to provide additional arguments to perform the refactoring (e.g., the new name of an element in the case of a rename refactoring).

5. The engine calls `checkFinalConditions` on the refactoring object. This is the most computationally intensive step. The refactoring object checks the preconditions of a refactoring, usually invoking several kinds of program analyses (e.g., data-flow analysis in case of ExtractMethod refactoring, type-constraints analysis [Tip07] in case of refactorings like ReplaceTypeWithTheMostGeneralSupertype). If the preconditions are not met, the engine warns the user. The user can decide to cancel the refactoring or may ignore the warnings and proceed (at her own risk) to the last step. A user should not ignore the warnings, although very rarely, due to the conservative nature of the static analyses, it might be safe to ignore the warnings.

6. The engine calls `createChange` method on the refactoring object. This is where the program transformation takes place. There are two different ways to implement the transformation: either make changes in the AST nodes and then pretty-print all AST nodes, or make changes in the AST nodes, compute the textual edits for each AST change, and apply the textual edits in the editor. The latter option is the preferred one since it preserves the original formatting of the source code, along with the comments, information that could otherwise get lost during the pretty-printing.

An important and useful feature in most refactoring engines is undo. The user can undo the transformation carried out by the refactoring. To support the undo feature, most refactoring engines represent refactorings as Command objects (see Command design pattern in [GHJV95]). A Command object has,

among others, two methods for undoing and redoing a certain action. In the case of refactoring command objects, the refactoring object needs to store enough information for the refactoring engine to undo/redo a refactoring. To expedite the execution of undo/redo, the refactoring engines stores the position in the file buffer and the edits caused by a refactoring during the last step (createChange). Thus, when undoing a refactoring, the engine efficiently does a textual replacement, rather than recomputing the preconditions (which is a potentially long-running operation).

3.3 Refactoring-aware Upgrading Theory

Upgrading can be regarded as another kind of program transformation (denoted later with a function μ). Before defining this transformation, we need to define three other functions on programs. We consider a program to be a set of program element declarations and use the same functions that Batory [Bat07] uses in the architectural metaprogramming literature:

- $(X + Y)$ is the set union of programs X and Y. " $+$ " is commutative, associative and has the empty program as the identity element.
- $(X - Y)$ is the set difference of sets X and Y (i.e., remove all terms Y from X). " $-$ " is not commutative, but is left associative.
- $R(X)$ which applies the transformation in refactoring R to program X, assuming that refactoring preconditions hold. If the refactoring preconditions hold for both the old and the new programs, applying the program transformation distributes over " $+$ " (i.e., $R(X + Y) = R(X) + R(Y)$) and over " $-$ " (i.e., $R(X - Y) = R(X) - R(Y)$).

As we saw in Section 3.1, refactorings can be composed. The composition of refactorings is denoted by ";" operator. For simplicity, we replace all individual refactorings with their composition, labeled R.

Initially, the program P is made up of the old version of the component and application:

$$P = Comp_{V_1} + App \tag{3.1}$$

The new version of the component contains refactorings, denoted by R, and other edits (e.g., bug fixes, performance improvements, new features), denoted by function E:

$$Comp_{V_2} = R(Comp_{V_1}) + E(Comp_{V_1}) \tag{3.2}$$

The upgraded program, P', is made up of the new version of the component and the refactored version of the application:

$$P' = Comp_{V_2} + R(App) \tag{3.3}$$

$$= R(Comp_{V_1}) + E(Comp_{V_1}) + R(App) \quad \text{[substitution using equation 3.2]} \tag{3.4}$$

Our solution to upgrade an application App from version V_1 of the component $Comp$ to version V_2 using refactoring technology can be summarized in these steps:

1. Recover the refactorings that changed component version V_1, $Comp_{V_1}$, into component version V_2, $Comp_{V_2}$. Let these refactorings be denoted R.
2. Reapply refactorings R on a larger program P that contains $Comp_{V_1}$ and App
3. Remove the refactored component from P and replace it with the new version of the component, $Comp_{V_2}$. This way, the application can benefit from other changes (e.g., bug fixes, performance improvements) in $Comp_{V_2}$ which are not refactorings.

Given the above notations and our upgrading process, we now define the upgrading transformation μ. The upgrading function takes program P and transforms it into P':

$\mu(P) = P'$

Definition 6 (Upgrading Function) *The function for upgrading a program X from $Comp_{V_1}$ to $Comp_{V_2}$, with respect to a sequence R of refactorings is:* $\mu(X) = R(X) - R(Comp_{V_1}) + Comp_{V_2}$

Applying this function to the program P consisting of $Comp_{V_1}$ and App results in:

$$
\begin{aligned}
\mu(P) &= \mu(Comp_{V_1} + App) && \text{[substitution 3.1]} \\
&= R(Comp_{V_1} + App) - R(Comp_{V_1}) + Comp_{V_2} && \text{[substitution using Definition 6]} \\
&= R(Comp_{V_1}) + R(App) - R(Comp_{V_1}) + Comp_{V_2} && \text{[using distributivity of refactorings]} \\
&= R(Comp_{V_1}) + R(App) - R(Comp_{V_1}) + R(Comp_{V_1}) + E(Comp_{V_1}) && \text{[substitution 3.2]} \\
&= R(Comp_{V_1}) + R(App) + E(Comp_{V_1}) \\
&= P' && \text{[substitution 3.3]}
\end{aligned}
$$

There are two observations to be made regarding the upgrade function. First, it assumes that most of the changes between two versions of a component are caused by refactorings. As we saw in Chapter 2, more than 80% of API-breaking changes in five real-world case studies were caused by refactorings. With respect to these changes, the upgrading function guarantees that the upgrade will not change the semantics of the upgraded application. However, because the term $Comp_{V_2}$ preserves the edits, these might change the semantics of the application.

Second, using the distributivity of refactorings over deletion, we can reduce the upgrading function, $\mu(X)$, to the following form:

$$\mu(X) = R(X - Comp_{V_1}) + Comp_{V_2} \tag{3.5}$$

Although the forms represented in equation 3.5 and definition 6 are equivalent, equation 3.5 seems to be more efficient, but it is not practical. Equation 3.5 upgrades a program X by first removing the component $Comp_{V_1}$, refactoring the remainder, and then adding component $Comp_{V_2}$. This seems to require less work than definition 6, since a tool has to refactor a smaller program, $X - Comp_{V_1}$. However, as we saw in the introduction to the refactoring engines, they all require that refactoring objects are initialized with the declaration of the program elements. Removing component $Comp_{V_1}$ before applying the refactorings would remove also the declarations of the program elements to be refactored, thus making it impossible for the refactoring engine to perform the transformation on the remainder of the program.

3.4 Practical Requirements for Refactoring-aware Upgrading Tools

To be practical, an upgrading tool based on refactorings needs to satisfy the needs of both application and component developers.

1. Component developers are reluctant to learn new annotation languages or write annotations/specifications whose sole purpose is to be used by upgrading tools. Informal interviews with the developers of the Eclipse framework reveal that component developers are not likely to write annotations extraneous to the regular component development. Therefore, upgrading tools need to automatically gather information about the component refactorings.

2. Application developers want an automated and safe (behavior-preserving) way to upgrade component-based applications to use the newer versions of components. Since refactorings are program transformations with well-defined semantics, an upgrading tool ought to take into account these semantics when integrating component refactorings into applications. In addition, the upgrading needs to preserve the edits in the component, since these edits might contain performance improvements and bug fixes. Eliminating component edits altogether would create a version of the component that contains only the structural changes, and therefore has no upgrading incentive for application developers.

First, with respect to meeting component developers' requirement, the upgrading tool needs to gather the refactorings that happened between two versions of a component (no overhead for component producers). Refactorings can be recorded right at the moment when they were performed on the component. This requires that the refactoring engine to automatically log each performed refactoring. Most refactoring tools, including Eclipse, represent refactorings as Command objects, but do not necessarily record all needed information to make these refactorings persistent. Thus, they will need to be changed to create a persistent log of refactorings. Alternatively, refactorings can be inferred automatically at a later time. This has the advantage that refactorings performed manually are inferred as well and included in the log of refactorings. Component developers distribute this log of refactorings along with the new release of the component.

Second, at the application's site, the upgrading tool needs to load the representation of the refactorings into live refactoring objects. If refactorings are already objects, then this might be as simple as reconstructing the object that had previously been stored.

Third, the upgrading tool incorporates the refactorings into applications. One possible solution is to replay the log of refactorings over the source code of the older version of the component and the application. Now the refactoring engine has access to the source code of the application as well, therefore all references to the refactored elements are correctly updated. During this step, the tool needs to handle those cases when a refactoring cannot be replayed in the new context. For example, at the component site it was possible to rename an API method, but at the application site this renaming results in a conflict because the application already defines a method having the new name.

3.5 Record and Replay

Next, we present a small extension to the refactoring engine that allows it to record-and-replay refactorings. Essentially, the refactoring engine can be viewed as a master "processor". This processor can be extended to invoke several slave "participants" during the refactoring process. One such participant can log information about a refactoring object once it is initialized.

To be able to replay the refactoring later, the recording participant needs to store information such as: what kind of refactoring is created, what is the program element on which it operates (the element is identified by its fully qualified name), what other parameters have been provided by the user through the UI (e.g., the new name in case of a rename refactoring or the default value of an argument in case of refactoring that changes a method signature by adding an argument). Out of this information, when replaying the refactoring, the extension needs to reconstitute a refactoring object and execute it. Execution entails invoking the precondition checking method (e.g., step 5 described in 3.2) and then the method that carries the transformation (e.g., step 6 described in 3.2).

We describe such an extension using Eclipse's refactoring engine. We collaborated to add this extension in the official release of Eclipse. In Eclipse, clients which contribute to refactoring history and refactoring scripting support need to register a subclass of `RefactoringContribution` with the extension point `org.eclipse.ltk.core.refactoring.refactoringContributions`. Refactoring contributions are stateless objects. They are instantiated on demand by the refactoring framework during the record or replay phase.

3.5.1 Recording Refactorings

Refactorings for which a refactoring contribution has been registered return a `RefactoringDescriptor` when their transformations are applied (step 6 described in subsection 3.2). This descriptor contains all the information required to replay the same refactoring later. It contains methods for serialization/deserialization into/from XML formats.

After a refactoring has been executed, the refactoring framework stores the returned refactoring descriptor into its global refactoring history. When creating the log of refactorings, the component developer can select which refactorings to distribute. For example, there might be refactorings that are local: they affect only private program elements in the component, but can not affect other applications.

Figure 3.4 shows the UI for creating such a log of all recorded refactorings. This log is saved in an XML format and can be distributed to clients along with the new version of the component.

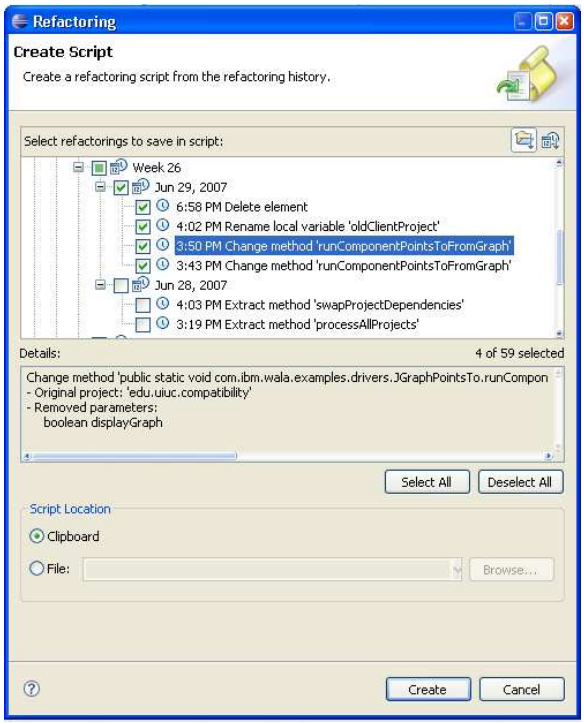

Figure 3.4: Creating a refactoring log script from the refactorings recorded in a project.

3.5.2 Replaying Refactorings

At the application site, the application developers load the file containing the refactoring logs and the refactoring engine replays it on the old version of the component and application. Since both the component and application reside on the same code base, the refactoring engine can correctly update all the references to the refactored component elements.

When a refactoring script is executed, the refactoring framework first retrieves the corresponding refactoring contribution for each kind of refactoring. This `RefactoringContribution` object knows how to deserialize refactoring descriptors stored in the log file. Next, the refactoring descriptor is used to dynamically construct the corresponding refactoring object and to initialize the refactoring object. The returned refactoring object is completely initialized and ready to be executed.

Figure 3.5 shows the UI for loading and executing a refactoring log file.

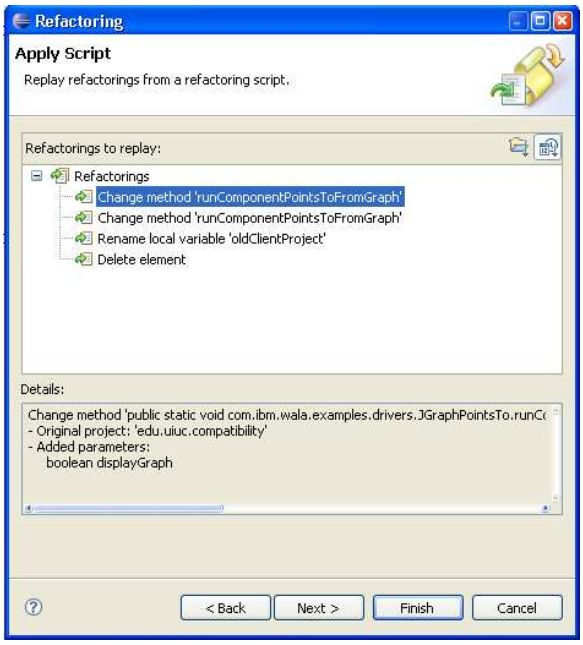

Figure 3.5: Applying a recorded refactoring script.

3.5.3 Limitations of Record-and-Replay

Although elegant and simple from an engineering point of view, the record-and-replay approach has three main limitations. This research addresses all these limitations.

1. First, replaying refactorings requires a log of refactorings. However, there is no log of recorded refac-

torings for the existing or legacy versions of the components. Also some refactorings are performed manually. To take full advantage of replay, it is crucial that refactorings can be automatically inferred. Chapter 4 presents our algorithm and tool for inferring refactorings.

2. Second, refactorings that are valid in the context of the component might be invalid in the context of component plus application. In addition, both component and application evolve not only through refactoring but also through edits. Refactorings and edits from component and application could interfere with each other, thus making impractical a naive replay of refactorings. Chapter 5 presents our solution to this problem: a refactoring-aware software merging algorithm.

3. Third, the source code of the application might not be available, or, because of legal reasons, can not be changed in response to component refactorings. Chapter 6 presents our solution for automatically generating a refactoring-aware compatibility layer that enables old applications to run along with the newer version of the component, without changing the applications.

3.6 Summary

This chapter introduces refactorings as program transformations and it gives a gentle introduction to refactoring engines. Then it presents our upgrading theory: component refactorings are replayed over the old version of the component and application, then the refactored version of the component is replaced with the last version of the component. This replacement preserves the edits in the last version of the component, while the replay incorporates the refactorings into the application. We present the practical requirements for a refactoring-aware upgrading tool and introduce a solution, record-and-replay that meets these requirements. We conclude by presenting the limitations of the record-and-replay and briefly describe what are the other parts of our toolset that address these limitations.

Chapter 4

Automated Detection of Refactorings

4.1 Introduction

One approach to automate the update of applications when their components change is to extend the refactoring engine to record refactorings on the component and then to replay them on the applications, as we saw in Section 3.5. While replay of refactorings shows great promise, it relies on the existence of refactoring logs. However, logs are not available for legacy versions of components. Also, logs will not be available for all future versions; some developers will not use refactoring engines with recording, and some developers will perform refactorings manually. To exploit the full potential of upgrading by replaying refactorings, it is therefore important to be able to automatically detect the refactorings used to create a new version of a component.

We propose a novel algorithm that detects a likely sequence of refactorings between two versions of a component. Previous algorithms [APM04, DDN00, GW05, GZ05, RD03] assumed closed-world development, where codebases are used only in-house and refactorings occur instantly (e.g., one entity dies in a version and a new refactored entity starts from the next version). However, in open-world development, components are reused outside the organization, therefore changes do not happen overnight but follow a long deprecate-replace-remove lifecycle. Obsolete entities will coexist with their newer counterparts until they are no longer supported. Also, multiple refactorings can happen to the same entity or related entities. This lifecycle makes it hard to accurately detect refactorings. Our algorithm works well for both closed- and open-world paradigms.

We want our algorithm to help the developer infer a log of refactorings for replay. To be practical, the algorithm needs to detect refactorings with a high accuracy. On one hand, if the algorithm adds to a log a change that is not actually a refactoring (false positive), the developer will need to remove it from the log or the replay could potentially introduce bugs. On the other hand, if the algorithm does not add to a log an

actual refactoring (false negative), the developer will need to manually find it and add it to the log.

In addition to detecting refactorings with high accuracy, our algorithm needs to scale up and analyze real world components containing hundreds of thousands of lines of code. Syntactic analyses scale up but are unreliable, while semantic analyses are more precise but are slow. To overcome these challenges, our algorithm combines a fast syntactic analysis to detect refactoring candidates and a precise semantic analysis to refine the results. Our syntactic analysis is based on Shingles encoding [Bro97], a technique from Information Retrieval. Shingles are a fast technique to find similar fragments in text files; our algorithm applies shingles to source files. Most refactorings involve repartitioning of the source files, which results in similar fragments of source text between different versions of a component. Our semantic analysis is based on the *reference graphs* that represent references among source-level entities, e.g., calls among methods[1]. This analysis considers the semantic relationship between candidate entities to determine whether they represent a refactoring.

We have implemented our algorithm as an Eclipse plugin, called RefactoringCrawler. RefactoringCrawler detects refactorings in Java components, although the ideas in the algorithm can be applied to other programming languages. RefactoringCrawler currently detects seven types of refactorings, focusing on refactorings that we found to be the most commonly applied in several components [DJ05]. We have evaluated RefactoringCrawler on three components ranging in size from 17 KLOC to 352 KLOC. The results show that RefactoringCrawler scales to real-world components, and its accuracy in detecting refactorings is over 85%.

RefactoringCrawler and our evaluation results are available on the website [Ref].

4.2 Example

We next illustrate some refactorings that our algorithm detects between two versions of a component. We use an example from the EclipseUI component of the Eclipse development platform. We consider two versions of EclipseUI, from Eclipse versions 2.1.3 and 3.0. Each of these versions of EclipseUI has over 1,000 classes and 10,000 methods in the public API (of non-internal packages). Our algorithm first uses a fast syntactic analysis to find similar methods, classes, and packages between the two versions of the component. (Section 4.4 presents the details of our syntactic analysis.) For EclipseUI, our algorithm finds

[1]These *references* do not refer to pointers between objects but to references among the source-code entities in each version of the component.

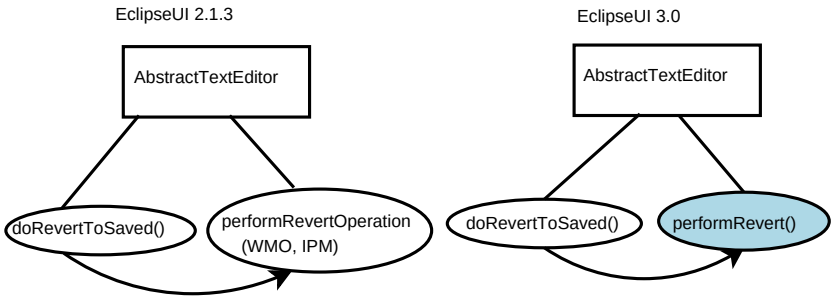

Figure 4.1: An excerpt from Eclipse versions 2.1 and 3.0 showing two refactorings, rename method and changed method signature, applied to the same method. The squares represent classes, the ellipses methods, and arrows are method calls. The method that changes signature also changes name from performRevert-Operation to performRevert.

231,453 pairs of methods with similar bodies, 487 pairs of similar classes, and 22 pairs of similar packages. (Section 4.8 presents more details of this case study.) These similar entities are candidates for refactorings. Our example focuses on two pairs of similar methods.

Figure 4.1 shows two pairs of similar methods from the two versions of the class `AbstractTextEditor` from Eclipse 2.1 and 3.0. The syntactic analysis finds that the method `doRevertToSaved` in version 2.1 is similar, although not identical, with method `doRevertToSaved` in version 3.0, and the method `performRevertOperation` is similar to the method `performRevert`. Our algorithm then uses a semantic analysis to detect the refactorings that were performed on these pairs. As a result, our algorithm detects that the method `performRevertOperation` was renamed to `performOperation`, and its signature changed from having two arguments in the version 2.1 to no argument in the version 3.0. Our previous manual inspection [DJ05] of the Eclipse documentation and code indeed found that these two refactorings, renamed method and changed method signature, were performed.

Our semantic analysis applies a series of detection strategies that find whether candidate pairs of similar entities are indeed results of refactorings. The key information that the strategies consider is the *references* between the entities in each version. For methods, the references correspond to call edges. For our example methods, both `performRevertOperation` and `performRevert` have only one call site in the entire EclipseUI: they are both called exactly once from `doRevertToSaved`. Our analysis represents this information with an edge, labeled with the number of calls, between these methods. We present how the two strategies for renamed methods and changed method signature proceed in our running example.

The RenameMethod strategy discards the pair of `doRevertToSaved` methods since they have the

same name. It then investigates whether `performRevert` is a renaming of `performRevertOperation`. The strategy finds the calls to these two methods and realizes that they are called (the same number of times) from the corresponding `doRevertToSaved` methods in both versions. Therefore, methods `performRevertOperation` and `performRevert` (i) are both in class `AbtstractTextEditor`, (ii) have similar method bodies, (iii) have similar incoming call edges, but (iv) differ in the name. The strategy concludes that `performRevert` is a renaming of `performRevertOperation`.

The ChangeMethodSignature strategy also considers pairs of similar methods. This strategy discards the pair of `doRevertToSaved` methods since they have the same signature. It then investigates `performRevertOperation` and `performRevert` methods, because they are now known represent the same method. It is important to point out here that strategies *share detected refactorings*: although `performRevertOperation` and `performRevert` seemingly have different names, the RenameMethod strategy has already found that these two methods correspond. The ChangedMethodSignature strategy then finds that `performRevertOperation` and `performRevert` (i) are in the same class, (ii) have similar method bodies, (iii) "same" name, (iv) similar call edges, but (v) different signatures. The strategy correctly concludes that a changed method signature refactoring was applied to `performRevert`.

4.3 Algorithm Overview

This section presents a high-level overview of our algorithm for detection of refactorings. Figure 4.2 shows the pseudo-code of the algorithm. The input are two versions of a component, and the output is a log of refactorings applied on `c1` to produce `c2`. The algorithm consists of two analyses: a fast *syntactic analysis* that finds candidates for refactorings and a precise *semantic analysis* that finds the actual refactorings.

Our syntactic analysis starts by parsing the source files of the two versions of the component into the *lightweight* ASTs, where the parsing stops at the declaration of the methods and fields in classes. For each component, the parsing produces a graph (more precisely, a tree to which analysis later adds more edges). Each node of the graphs represents a source-level entity, namely a package, a class, a method, or a field. Each node stores a fully qualified name for the entity, and each method node also stores the fully qualified names of method arguments to distinguish overloaded methods. Nodes are arranged hierarchically in the tree, based on their fully qualified names: the node $p.n$ is a child of the node p.

The heart of our syntactic analysis is the use of the *Shingles encoding* to find similar pairs of enti-

```
Refactorings detectRefactorings(Component c1, c2) {
  // syntactic analysis
  Graph g1 = parseLightweight(c1);
  Graph g2 = parseLightweight(c2);
  Shingles s1 = annotateGraphNodesWithShingles(g1);
  Shingles s2 = annotateGraphNodesWithShingles(g2);
  Pairs pairs = findSimilarEntities(s1, s2);
  // semantic analysis
  Refactorings rlog = emptyRefactorings();
  foreach (DetectionStrategy strategy) {
    do {
     Refactorings rlog' = rlog.copy();
     foreach (Pair<e1, e2> from pairs relevant to strategy)
        if (strategy.isLikelyRefactoring(e1, e2, rlog))
          rlog.add(<e1, e2>, strategy);
    } while (!rlog'.equals(rlog)); // fixed point
  }
  return rlog;
}
```

Figure 4.2: Pseudo-code of the conceptual algorithm for detection of refactorings.

ties (methods, classes, and packages) in the two versions of the component. Shingles are "fingerprints" for strings with the following property: if a string changes slightly, then its shingles also change slightly. Therefore, shingles enable detection of strings with similar fragments much more robustly than the traditional string matching techniques that are sensitive to small perturbations like renamings of local variables or small edits. Section 4.4 presents the computation of shingles in detail.

The result of our syntactic analysis is a set of pairs of entities that have similar shingles encodings in the two versions of the component. Each pair consists of an entity from the first version and an entity of the same kind from the second version; there are separate sets for methods, classes, and packages. Pairs indicate candidate refactorings.

Our semantic analysis detects from the candidate pairs those where the second entity is a likely refactoring of the first entity. The analysis applies seven strategies for detecting specific refactorings, such as RenameMethod or ChangeMethodSignature discussed in the motivation example in Section 4.2. Section 4.5 presents the strategies in detail. The analysis applies each strategy until it finds all possible refactorings of its type. Each strategy considers all pairs of entities $\langle e_1, e_2 \rangle$ of the appropriate type, e.g., RenameMethod considers only pairs of methods. For each pair, the strategy computes how likely is that e_1 was refactored into e_2; if the likelihood is above a user-specified threshold, the strategy adds the pair to the log of refactorings that the subsequent strategies can use during further analysis. Note that each strategy takes into

account already detected refactorings; sharing detected refactorings among strategies is a key for accurate detection of refactorings when multiple types of refactorings applied to the same entity (e.g., a method was renamed and has a different signature) or related entities (e.g., a method was renamed and also its class was renamed). Our analysis cannot recover the list of refactorings in the order they were performed, but it finds *one path* that leads to the same result.

4.4 Syntactic Analysis

To identify possible candidates for refactorings, our algorithm first determines pairs of *similar* methods, classes, and packages. Our algorithm uses the Shingles encoding [Bro97] to compute a fingerprint for each method and determines two methods to be similar if and only if they have similar fingerprints. Unlike the traditional hashing functions that map even the smallest change in the input to a completely different hash value, the Shingles algorithm maps small changes in the input to small changes in the fingerprint encoding.

4.4.1 Computing Shingles for Methods

The Shingles algorithm takes as input a sequence of tokens and computes a multiset of integers called shingles. The tokens represent the method body or the Javadoc comments for the method (as interface methods and abstract methods have no body). The tokens do not include method name and signature because refactorings affect these parts. The algorithm takes two parameters, the length of the sliding window, W, and the maximum size of the resulting multiset, S. Given a sequence of tokens, the algorithm uses the sliding window to find all subsequences of length W, computes the shingle for each subsequence, and selects the S minimum shingles for the resulting multiset. Instead of selecting S shingles which have minimum values, the algorithm could use any other heuristic that deterministically selects S values from a larger set. Our implementation uses the Rabin's hash function [Rab81] to compute the shingles.

If the method is short and has fewer than S shingles, then the multiset contains all shingles. This is the case with many setters and getters and some constructors and other initializers. The parameter S acts as the upper bound for the space needed to represent shingles: a larger value of S makes calculations more expensive, and a smaller value makes it harder to distinguish strings. Our implementation sets the number of shingles proportional to the length of the method body/comments.

Figure 4.3 shows the result of calculating the shingles for two method bodies with $W = 2$ and $S = 10$.

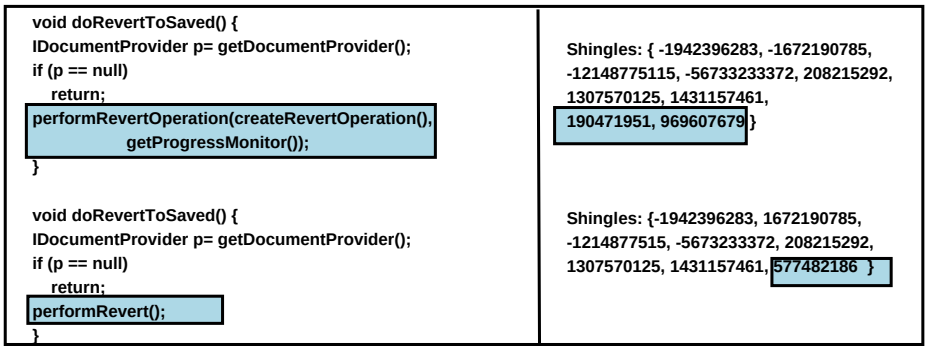

Figure 4.3: Shingles encoding for two versions of `AbstractTextEditor.doRevertToSaved` between Eclipse 2.1 and 3.0. Notice that small changes (gray boxes) in the input strings produce small changes in the Shingles encoding.

The differences in the bodies and the shingle values are in grey boxes. Notice that the small changes in the tokens produce only small changes in the shingle representation, enabling the algorithm to find the similarities between methods.

4.4.2 Computing Shingles for Classes and Packages

The shingles for methods are used to compute shingles for classes and packages. The shingles for a class are the minimum S_{class} values of the union of the shingles of the methods in that class. Analogously, the shingles for a package are the minimum $S_{package}$ values of the union of the shingles of the classes in that package. This way, the algorithm efficiently computes shingles values and avoids recalculations.

4.4.3 Finding Candidates

Our analysis uses the shingles to find candidates for refactorings. Each candidate is a pair of similar entities from the two versions of the component. This analysis is an effective way of eliminating a large number of pairs of entities, so that the expensive operation of computing the reference graphs is only done for a small subset of all possible pairs. More specifically, let M_1 and M_2 be the multisets of shingles for two methods, classes, or packages. Our analysis computes similarity between these two multisets. Let $|M_1 \cap M_2|$ be the cardinality of the intersection of M_1 and M_2. To compare similarity for different pairs, the algorithm *normalizes* the similarity to be between 0 and 1. More precisely, the algorithm computes the similarity as the *average* of similarity from M_1 to M_2 and similarity from M_2 to M_1 to address the cases when M_1 is

similar to M_2 but M_2 is not similar to M_1 :

$$\frac{\frac{|M_1 \cap M_2|}{|M_1|} + \frac{|M_2 \cap M_1|}{|M_2|}}{2}.$$

If this similarity value is above the user-specified threshold, the pair is deemed similar and passed to the semantic analysis.

4.5 Semantic Analysis

We present the semantic analysis that our algorithm uses to detect refactorings. Recall from Figure 4.2 that the algorithm applies each detection strategy until it reaches a fixed point and that all strategies share the same log of detected refactorings, `rlog`. This sharing is crucial for successful detection of refactorings when multiple types of refactorings happened to the same entity (e.g., a method was renamed and has a different signature) or related entities (e.g., a method was renamed and also its class was renamed). We first describe how the strategies use the shared log of refactorings. We then describe *references* that several strategies use to compute the likelihood of refactoring. We also define the multiplicity of references and the similarity that our algorithm computes between references. We finally presents details of each strategy. Due to the sharing of the log, our algorithm imposes an order on the types of refactorings it detects first. Specifically, the algorithm applies the strategies in the following order:

1. RenamePackage (RP)
2. RenameClass (RC)
3. RenameMethod (RM)
4. PullUpMethod (PUM)
5. PushDownMethod (PDM)
6. MoveMethod (MM)
7. ChangeMethodSignature (CMS)

4.5.1 Shared Log

The strategies compare whether an entity in one graph corresponds to an entity in another graph *with respect to the already detected refactorings*, in particular with renaming refactorings. Suppose that the refactorings log `rlog` already contains several renamings that map fully qualified names from version `c1` to version `c2`. These renamings map package names to package names, class names to class names, or method names to method names. We define a renaming function ρ that maps a fully qualified name `fqn` from `c1` with respect to the renamings in `rlog`:

$$\begin{aligned} \rho(\texttt{fqn}, \texttt{rlog}) &= \text{if (defined } \texttt{rlog}(\texttt{fqn})) \text{ then } \texttt{rlog}(\texttt{fqn}) \\ &\quad \text{else } \rho(\text{pre}(\texttt{fqn}), \texttt{rlog}) + \texttt{"."} + \text{suf}(\texttt{fqn}) \\ \rho(\texttt{""}, \texttt{rlog}) &= \texttt{""}, \end{aligned}$$

where suf and pre are functions that take a fully qualified name and return its simple name (*suffix*) and the entire name but the simple name (*prefix*), respectively. The function ρ recursively checks whether a renaming of some part of the fully qualified name is already in `rlog`.

4.5.2 References

The strategies compute the likelihood of refactoring based on *references* among the source-code entities in each of the two versions of the component. In each graph that represents a version of the component, our algorithm (lazily) adds an edge from a node n' to a node n if the source entity represented by n' has a reference to a source entity represented by n. (The graph also contains the edges from the parse tree.) We define references for each kind of nodes/entities in the following way:

- There is a reference from a node/method m' to a node/method m iff m' calls m. Effectively, references between methods correspond to the edges in call graphs.
- There is a reference from a node n' to a node/class C iff:
  - n' is a method that has (i) an argument or return of type C, or (ii) an instantiation of class C, or (iii) a local variable of class C.

– n' is a class that (i) has a field whose type is C or (ii) is a subclass of C.

- There is a reference from a node n' to a node/package p iff n' is a class that imports some class from the package p.

There can be several references from one entity to another. For example, one method can have several calls to another method or one class can have several fields whose type is another class. Our algorithm assigns to each edge a *multiplicity* that is the number of references. For example, if a method m' has two calls to a method m, then the edge from the node n' that represents m' to the node n that represents m has multiplicity two. Conceptually, we consider that there is an edge between any two nodes, potentially with multiplicity zero. We write $\mu(n', n)$ for the multiplicity from the node n' to the node n.

4.5.3 Similarity of References

Our algorithm uses a metric to determine the similarity of references to entities in the two versions of the component, with respect to a given log of refactorings. We write $n \in \mathtt{g}$ for a node n that belongs to a graph $\mathtt{g}$. Consider two nodes $n_1 \in \mathtt{g1}$ and $n_2 \in \mathtt{g2}$. We define the similarity of their incoming edges as follows. We first define the *directed similarity* between two nodes with respect to the refactorings. We then take the overall similarity between n_1 and n_2 as the average of directed similarities between n_1 and n_2 and between n_2 and n_1. The average of directed similarities helps to compute a fair grade when n_1 is similar to n_2 but n_2 is not similar to n_1.

We define the directed similarity between two nodes n and n' as the overlap of multiplicities of their *corresponding* incoming edges. More precisely, for each incoming edge from a node n_i to n, the directed similarity finds a node $n_i' = \rho(n_i, \mathtt{rlog})$ that corresponds to n_i (with respect to refactorings) and then computes the overlap of multiplicities between the edges from n_i to n and from n_i' to n'. The number of overlapping incoming edges is divided by the total number of incoming edges. The formula for directed similarity is:

$$\delta(n, n', \mathtt{rlog}) = \frac{\sum_{n_i} \min(\mu(n_i, n), \mu(\rho(n_i, \mathtt{rlog}), n'))}{\sum_{n_i} \mu(n_i, n)}$$

The overall similarity is the average of directed similarities:

$$\sigma(n_1, n_2, \texttt{rlog}) = \frac{\delta(n_1, n_2, \texttt{rlog}) + \delta(n_2, n_1, \texttt{rlog}^{-1})}{2}$$

When computing the directed similarity between n_2 and n_1, the algorithm needs to take into account the inverse of renaming log, denoted by $\texttt{rlog}^{-1}$. Namely, starting from a node n_i in g_2, the analysis searches for a node $n_{i'}$ in g_1 such that the renaming of $n_{i'}$ (with respect to `rlog`) is n_i, or equivalently, $\rho(n_i, \texttt{rlog}^{-1}) = n_{i'}$.

We describe informally an equivalent definition of directed similarity based on the view of graphs with multiplicities as multigraphs that can have several edges between two same nodes. The set of edges between two nodes can be viewed as a multiset, and finding the overlap corresponds to finding the intersection of one multiset of edges with the other multiset of edges (for nodes corresponding with respect to the refactorings). In this view, similarity between edges in the graph is conceptually analogous to the similarity of multisets of shingles.

4.5.4 Detection Strategies

Next we describe all detection strategies for refactorings. Each strategy checks appropriate pairs of entities and has access to the graphs `g1` (corresponding to the old version of the component), `g2` (corresponding to the newer version), and the `rlog` of refactorings. (See the call to `isLikelyRefactoring` in Figure 4.2.) Figure 4.4 shows the seven strategies currently implemented in RefactoringCrawler. For each pair, the strategy first performs a fast syntactic check that determines whether the pair is relevant for the refactoring and then performs a semantic check that determines the likelihood of the refactoring. The semantic checks compare the similarity of references to the user-specified threshold value T.

RenamePackage (RP), RenameClass (RC), and RenameMethod (RM) strategies are similar. The first syntactic check requires the entity from `g2` not to be in `g1`; otherwise, the entity is not new. The second check requires the two entities to have the same name prefix, modulo the renamings in `rlog`; otherwise, the refactoring is a potential move but not a rename. The third check requires the two entities to have different simple names.

PullUpMethod (PUM) and PushDownMethod (PDM) are the opposite of each other. Figure 4.5 illustrates a PUM that pulls up the declaration of a method from a subclass into the superclass such that the method can be reused by other subclasses. Figure 4.6 illustrates a PDM that pushes down the declaration

Refactoring	Syntactic Checks	Semantic Checks
RP(p_1, p_2)	$p_2 \notin$ `g1` ρ(pre(p_1), `rlog`) = pre(p_2) suf(p_1) $\neq$ suf(p_2)	$\sigma(p_1, p_2,$ `rlog`$) \geq$ T
RC(C_1, C_2)	$C_2 \notin$ `g1` ρ(pre(C_1), `rlog`) = pre(C_2) suf(C_1) $\neq$ suf(C_2)	$\sigma(C_1, C_2,$ `rlog`$) \geq$ T
RM(m_1, m_2)	$m_2 \notin$ `g1` ρ(pre(m_1), `rlog`) = pre(m_2) suf(m_1) $\neq$ suf(m_2)	$\sigma(m_1, m_2,$ `rlog`$) \geq$ T
PUM(m_1, m_2)	$m_2 \notin$ `g1` ρ(pre(m_1), `rlog`) $\neq$ pre(m_2) suf(m_1) = suf(m_2)	$\sigma(m_1, m_2,$ `rlog`$) \geq$ T ρ(pre(m_1), `rlog`) descendant-of pre(m_2)
PDM(m_1, m_2)	$m_2 \notin$ `g1` ρ(pre(m_1), `rlog`) $\neq$ pre(m_2) suf(m_1) = suf(m_2)	$\sigma(m_1, m_2,$ `rlog`$) \geq$ T ρ(pre(m_1), `rlog`) ancestor-of pre(m_2)
MM(m_1, m_2)	$m_2 \notin$ `g1` ρ(pre(m_1), `rlog`) $\neq$ pre(m_2) suf(m_1) = suf(m_2)	$\sigma(m_1, m_2,$ `rlog`$) \geq$ T $\neg\rho$(pre(m_1), `rlog`) anc.-or-desc. pre(m_2) references-properly-updated
CMS(m_1, m_2)	ρ(fqn(m_1), `rlog`) = fqn(m_2) signature(m_1) $\neq$ signature(m_2)	$\sigma(m_1, m_2,$ `rlog`$) \geq$ T

Figure 4.4: Syntactic and semantic checks performed by different detection strategies for refactorings: RP=RenamePackage, RC=RenameClass, RM=RenameMethod, PUM=PullUpMethod, PDM=PushDownMethod, MM=MoveMethod, and CMS=ChangeMethodSignature.

of a method from a superclass into a subclass that uses the method because the method is no longer reused by other subclasses. In general, the PUM and PDM can be between several classes related by inheritance, not just between the immediate subclass and superclass; therefore, PUM and PDM check that the original class is a *descendant*, respectively an *ancestor*, of the target class. These inheritance checks are done on the graph `g2`.

MoveMethod (MM) has the second syntactic check that requires the parent classes of the two methods to be different. Without this check, MM would incorrectly classify all methods of a renamed class as moved methods. The second semantic check requires that the declaration classes of the methods not be related by inheritance; otherwise, the refactorings would be incorrectly classified as MM as opposed to a PUM/PDM. The third check requires that all references to the target class be removed in the second version and that all calls to methods from the initial class be replaced with sending a message to an instance of the initial class. We illustrate this check on the sample code in Figure 4.7. In the first version, method `C1.m1` calls a method `C1.xyz` of the same class `C1` and also calls a method `C2.m2`. After `m1` is moved to the class `C2`, `m1` can call

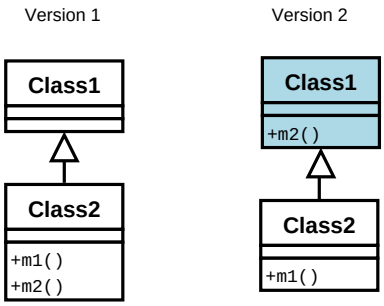

Figure 4.5: PullUpMethod: method `m2` is pulled up from the subclass `C2` into the superclass `C1`.

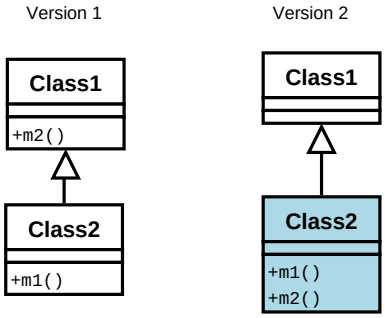

Figure 4.6: PushDown: method `m2` is pushed down from the superclass `C1` into the subclass `C2`.

any method in `C2` directly (e.g., `m2`), but any calls to methods residing in `C1` need to be executed through an instance of `C1`.

ChangeMethodSignature (CMS) looks for methods that have the same fully qualified name (modulo renamings) but different signatures. The signature of the method can change by gaining/loosing arguments, by changing the type of the arguments, by changing the order of the arguments, or by changing the return type.

4.6 Discussion of the Algorithm

The example from Section 4.2 illustrates some of the challenges in automatic detection of refactorings that happened in reusable components. We next explicitly discuss three main challenges and present how our algorithm addresses them.

The first challenge is the size of the code to be analyzed. An expensive semantic analysis—for example finding similar subgraphs in call graphs (more generally, in the entire reference graphs)—might detect refactorings but does not scale up to the size of real-world components with tens of thousands of entities,

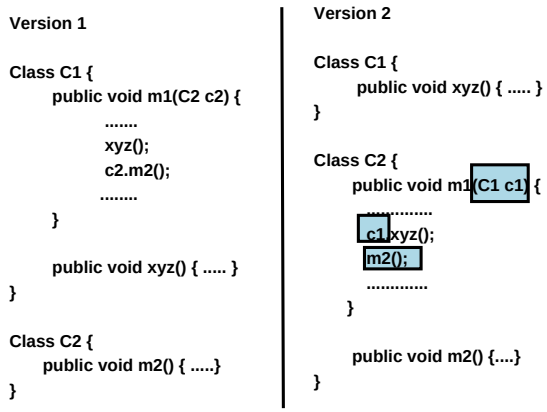

Figure 4.7: Method `m1` moves from class `C1` in one version to class `C2` in the next version. The method body changes to reflect that the local methods (e.g., `m2`) are called directly, while methods from the previous class (e.g., `xyz`) are called indirectly through an instance of `C1`.

including methods, classes, and packages. A cheap syntactic analysis, in contrast, might find many similar entities but is fallible to renamings. Also, an analysis that would not take into account the semantics of entity relationships would produce a large number of false positives. Our algorithm uses a hybrid of syntactic and semantic analyses: a fast syntactic analysis creates pairs of candidate entities that are suspected of refactoring, and a more precise semantic analysis on these candidates detects whether they are indeed refactorings.

The second challenge is the noise introduced by preserving backward compatibility in the components. Consider for example the following change in the Struts framework from version 1.1 to version 1.2.4: the method `perform` in the class `Controller` was renamed to `execute`, but `perform` still exists in the later version. However, `perform` is deprecated, all the internal references to it were replaced with references to `execute`, and the users are warned to use `execute` instead of `perform`. Since it is not feasible to perform an expensive analysis on all possible pairs of entities across two versions of a component, any detection algorithm has to consider only a subset of pairs. Some previous algorithms [APM04, DDN00, GZ05] consider only the entities that "die" in one version and then search for refactored counterparts that are created in the next version. The assumption that entities change in this fashion indeed holds in the closed-world development (where the only users of components are the component developers) but does not hold in the open-world development where obsolete entities coexist with their refactored counterparts. For example, the previous algorithms cannot detect that `perform` was renamed to `execute` since `perform` still exists in the subsequent version. Our algorithm detects that `perform` in the first version and `execute`

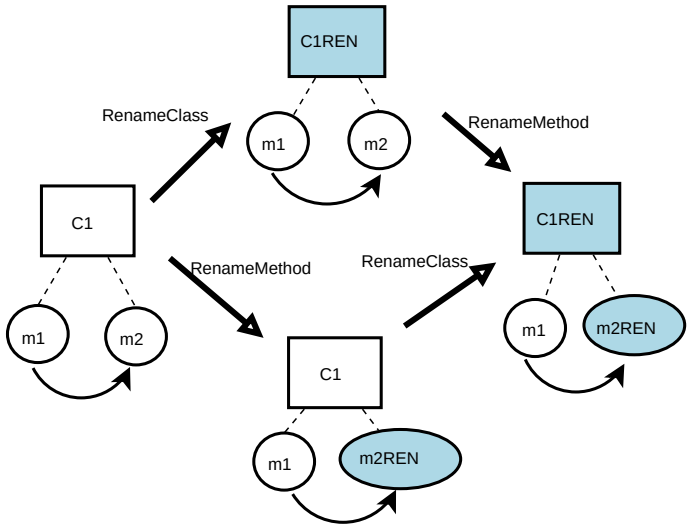

Figure 4.8: Two refactorings affect related entities class $C1$ and method $m2$. The class rename happens before the method rename in the upper path, the reverse happens in the bottom path. Both paths end up with the same result.

in the second version have the same shingles and their call sites are the same, and therefore our algorithm correctly classifies the change as a method rename.

The third challenge is multiple refactorings happening to the same entity or related entities. The example from Section 4.2, for instance, shows two refactorings, rename method and change method signature, applied to the same method. An example of refactorings happening to related entities is renaming a method along with renaming the method's class. Figure 4.8 illustrates this scenario. Across the two versions of a component, class `C1` was renamed to `C1REN`, and one of its methods, `m2`, was renamed to `m2REN`. During component evolution, regardless of whether the class or method rename was executed first, the end result is the same. In Figure 4.8, the upper part shows the case when the class rename was executed first, and the lower part shows the case when the method rename was executed first.

Our algorithm addresses the third challenge by imposing an order on the detection strategies and sharing the information about detected refactorings among the detection strategies. Any algorithm that detects refactorings conceptually reconstructs the log of refactorings and thus not only the start and the end state of a component but also the intermediate states. Our algorithm detects the two refactorings in Figure 4.8 by following the upper path. When detecting a class rename, the algorithm takes into account only the shingles for class methods and not the method names. Therefore, our algorithm detects class `C1REN` as a rename of class `C1` although one of its methods was renamed. This information is fed back into the loop; it

conceptually reconstructs the state 2a, and the analysis continues. The subsequent analysis for the rename method checks whether the new-name method belongs to the same class as the old-name method; since the previous detection discovered that `C1` is equivalent modulo rename with `C1REN`, `m2REN` can be detected as a rename of `m2`.

The order in which an algorithm detects the two refactorings matters. We described how our algorithm detects a class rename followed by a method rename. Consider, in contrast, what would happen to an algorithm that attempts to follow the bottom path. When analyzing what happened between the methods `m2` and `m2REN`, the algorithm would need the intermediate state 2b (where `m2REN` belongs to `C1`) to detect that `m2` was renamed to `m2REN`. However, that state is not given, and in the end state `m2REN` belongs to `C1REN`, so the algorithm would mistakenly conclude that `m2REN` was moved to another class (`C1REN`). The subsequent analysis of what happened between classes `C1` and `C1REN` would presumably find that they are a rename and would then need to backtrack to correct the previously misqualified move method as a rename method. For this reason, our algorithm imposes an order on the detection strategies and runs detection of renamings top-down, from packages to classes to methods.

To achieve a high level of accuracy, our algorithm uses a fixed-point computation in addition to the ordering of detection strategies. The algorithm runs each strategy repeatedly until it finds no new refactorings. This loop is necessary because entities are intertwined with other entities, and a strategy cannot detect a refactoring in one entity until it detects a refactoring in the dependent entities. For instance, consider this example from Struts framework versions 1.1 and 1.2.4: in the class `ActionController`, the method `perform` was renamed to `execute`. The implementation of `perform` in `ActionController` is a utility class that merely delegates to different subclasses of `Action` by sending them a `perform` message. For 11 of these `Action` classes, their callers consist mostly of the `ActionController.perform`. Therefore, unless a tool detects first that `perform` was renamed to `execute`, it cannot detect correctly the similarity of the incoming call edges for the other 11 methods. After the first run of the RenameMethod detection, our RefactoringCrawler tool misses the 11 other method renames. However, the feedback loop adds the information about the rename of `perform`, and the second run of the RenameMethod detection correctly finds another 11 renamed methods.

Even though we only analyze seven types of refactorings, conceptually similar combination of syntactic and semantic analysis can detect many other types of refactorings. A lot of the refactorings published by

Fowler et al. [FBB+99] can be detected in this way, including extract/inline method, extract/inline package, extract/inline class or interface, move class to different package, collapse class hierarchy into a single class, replace record with data class, replace anonymous with nested class, replace type conditional code with polymorphism, as well as some higher-level refactorings to design patterns [GHJV95] including create Factory methods, form Template Method, replace type code with State/Strategy.

The largest extension to the current algorithm is required by 'replace type conditional code with polymorphism'. This refactoring replaces a switch statement whose branches type-check the exact type of an object (e.g., using *instanceof* in Java) with a call to a polymorphic method that is dynamically dispatched to the right class at run time. All the code in each branch statement is moved to the class whose type was checked in that branch. To detect this refactoring, the syntactic analysis should not only detect similar methods, but also similar statements and expressions within method bodies. This requires that shingles are computed for individual statements and expressions, which is overhead to the current implementation, but offers a finer level of granularity. Upon detection of similar statements in a switch branch and in a class method, the semantic analysis needs to check whether the class has the same type as the one checked in the branch and whether the switch is replaced in the second version with a call to the polymorphic method.

4.7 Implementation

We have implemented our algorithm for detecting refactorings in RefactoringCrawler, a plugin for the Eclipse development environment. The user loads the two versions of the component to be compared as projects inside the Eclipse workspace and selects the two projects for which RefactoringCrawler detects refactorings. To experiment with the accuracy and performance of the analysis, the user can set the values for different parameters, such as the size of the sliding window for the Shingles encoding (Section 4.4); the number of shingles to represent the digital fingerprint of methods, classes and package; and the thresholds used in computing the similarity of shingles encoding or the reference graphs. RefactoringCrawler provides a set of default parameter values that should work fine for most Java components.

RefactoringCrawler provides an efficient implementation of the algorithm shown in Figure 4.2. The syntactic analysis starts by parsing the source files of the two versions of the component and creates a graph representation mirroring the *lightweight* ASTs. We call it lightweight because the parsing stops at the declaration of the methods and fields in classes. RefactoringCrawler then annotates each method and field

node with shingles values corresponding to the source code behind each node (e.g. method body or field initializers). From the leaves' shingles values, RefactoringCrawler annotates (bottom-up) with shingles values all the nodes corresponding to classes and packages. Since each node contains the fully qualified name of the source code entity, it is easy to navigate back and forth between the actual source code and the graph representation.

During the semantic analysis, RefactoringCrawler uses Eclipse's search engine to find the references among source code entities. The search engine operates on the source code, not on the graph. The search engine does a type analysis to identify the class of a reference when two methods in unrelated classes have the same name. Finding the references is an expensive computation, so RefactoringCrawler lazily runs this and caches the intermediate results by adding edges between the graph nodes that refer each other.

RefactoringCrawler performs the analysis and returns back the results inside an Eclipse view. RefactoringCrawler presents only the refactorings that happened to the public API level of the component since only these can affect the component users. RefactoringCrawler groups the results in categories corresponding to each refactoring strategy. Double clicking on any leaf Java element opens an editor having selected the declaration of that particular Java element. RefactoringCrawler also allows the user to export the results into an XML format compatible with the format that Eclipse uses to load a log of refactorings. Additionally, the XML format allows the developer to further analyze and edit the log, removing false positives or adding missed refactorings.

The reader can see screenshots and is encouraged to download the tool from the website [Ref].

4.8 Evaluation

We evaluate RefactoringCrawler on three real-world components. To measure the accuracy of RefactoringCrawler, we need to know the refactorings that were applied in the components. Therefore, we chose the components from our previous study [DJ05] that analyzed the API changes in software evolution and found refactorings to be responsible for more than 80% of the changes. The previous study considered components with good release notes describing the API changes. Starting from the release notes, we manually discovered the refactorings applied in these components. These manually discovered refactorings helped us to measure the accuracy of the refactoring logs that RefactoringCrawler reports. In general, it is easier to detect the false positives (refactorings that RefactoringCrawler erroneously reports) by comparing the reported refac-

	Size KLOC	Packages	Classes	Methods	ReleaseNotes [Pages]
Eclipse.UI 2.1.3	222	105	1151	10285	-
Eclipse.UI 3.0	352	192	1735	15894	8
Struts 1.1	114	88	460	5916	-
Struts 1.2.4	97	78	469	6044	16
JHotDraw 5.2	17	19	160	1458	-
JHotDraw 5.3	27	19	195	2038	3

Table 4.1: The size of components used as case studies.

torings against the source code than it is to detect the false negatives (refactorings that RefactoringCrawler misses). To determine false negatives, we compare the manually found refactorings against the refactorings reported by RefactoringCrawler. Additionally, RefactoringCrawler found a few refactorings that were not documented in the release notes. Our previous study and the evaluation of RefactoringCrawler allowed us to build a repository of refactorings that happened between the two versions of the three components. The case studies along with the tool and the detected refactorings can be found online [Ref].

For each component, we need to choose two versions. The previous study [DJ05] chose two major releases that span large architectural changes because such releases are likely to have lots of changes and to have the changes documented. We use the same versions to evaluate RefactoringCrawler. Note, however, that these versions can present hard cases for RefactoringCrawler because they are far apart and can have large changes. RefactoringCrawler still achieves practical accuracy for these versions. We believe that RefactoringCrawler could achieve even higher accuracy on closer versions with less changes.

4.8.1 Case Study Components

Table 4.1 shows the size of the case study components. ReleaseNotes give the size (in pages) of the documents that the component developers provided to describe the API changes. We described these case study components in Section 2.2.1.

4.8.2 Measuring the Recall and Precision

To measure the accuracy of RefactoringCrawler, we use precision and recall, two standard metrics from the Information Retrieval field. *Precision* tells how many of the refactorings reported by the tool are genuine refactorings. Formally, it is the ratio of the number of relevant refactorings found by the tool to the total

	RM	*RC*	*RP*	*MM*	*PUM*	*PDM*	*CMS*	*Precision*	*Recall*
EclipseUI 2.1.3 - 3.0	2,1,0	0,0,0	0,0,0	8,2,4	11,0,0	0,0,0	6,0,0	90%	86%
Struts 1.2.1 - 1.2.4	20,0,1	1,0,1	0,0,0	20,0,7	1,0,0	0,0,0	24,0,1	100%	86%
JHotDraw 5.2 - 5.3	5,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	19,0,0	100%	100%

Table 4.2: Triples of (GoodResults, FalsePositives, FalseNegatives) for RenameMethod(RM), RenameClass(RC), RenamePackage(RP), MoveMethod(MM), PullUpMethod(PUM), PushDownMethod(PDM), ChangeMethodSignature(CMS).

number of irrelevant (spurious, false positives) and relevant refactorings found by the tool. It is expressed as the percentage:

$$PRECISION = GoodResults/(GoodResults + FalsePositives)$$

Recall tells how many of the refactorings that the tool should have found it actually found. Formally, it is the ratio of the number of genuine refactorings found by the tool (good results) to the total number of actual refactorings in the component. It is expressed as the percentage:

$$RECALL = GoodResults/(GoodResults + FalseNegatives)$$

Ideally, precision and recall should be 100%. If that was the case, the reported refactorings could be fed directly into a tool that replays them to automatically upgrade component-based applications. However, due to the challenges mentioned in Section 4.6, it is hard to have 100% precision and recall.

Table 4.2 shows how many instances of each refactoring were found for the three components. These results use the default values for the parameters in RefactoringCrawler [Ref]. For each refactoring type, we show in a triple how many good results RefactoringCrawler found, how many false positives RefactoringCrawler found, and how many refactorings it missed (false negatives are obtained from the release notes [DJ05]). For each component, we compute the overall precision and recall by taking into account all seven kinds of refactorings.

We further analyzed why RefactoringCrawler missed a few refactorings. In Struts, for instance, method `RequestUtils.computeParameters` is moved to `TagUtils.computeParameters`, and method `RequestUtils.pageURL` is moved to `TagUtils.pageURL`. There are numerous calls to these methods from a test class. However, it appears that the test code was not refactored, and therefore

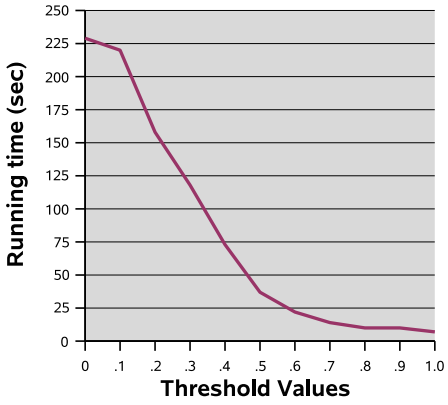

Figure 4.9: Running time for JHotDraw decreases exponentially with higher threshold values used in the syntactic analysis.

it still calls the old method (which is deprecated). This results in quite different call sites for the old and the refactored method, thus misleading the semantic analysis.

4.8.3 Performance

The results in Table 4.2 were obtained when RefactoringCrawler ran on a Fujitsu laptop with a 1.73GHz Pentium 4M CPU and 1.25GB of RAM. It took 16 min 38 sec for detecting the refactorings in EclipseUI, 4 min and 55 sec for Struts, and 37 sec for JHotDraw. Figure 4.9 shows how the running time for JHotDraw varies with the change of the method similarity threshold values used in the syntactic analysis. For low threshold values, the number of candidate pairs passed to the semantic analysis is large, resulting in longer analysis time. For high threshold values, fewer candidate pairs pass into the semantic analysis, resulting in lower running times. For JHotDraw, a .1 method similarity threshold passes 1842 method candidates to the RenameMethod's semantic analysis, a .5 threshold value passes 88 candidates, while a .9 threshold passes only 4 candidates.

The more important question, however, is how precision and recall vary with the change of the similarity threshold values. Very low threshold values produce a larger number of candidates to be analyzed, which results in a larger number of false positives, but increases the chance that all the relevant refactorings are found among the results. Very high threshold values imply that only those candidates that have almost perfect body resemblance are taken into account, which reduces the number of false positives but can miss some refactorings. We have found that threshold values between 0.5 and 0.7 result in practical precision and

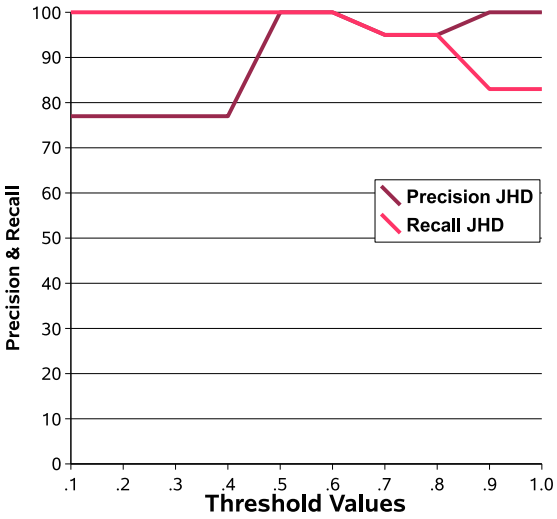

Figure 4.10: Recall and Precision vary with the value of similarity threshold for JHotDraw case study.

recall. Figure 4.10 and 4.11 show how the precision and recall vary for different similarity thresholds for JHotDraw, respectively Struts case study.

4.8.4 Strengths and Limitations

We next discuss the strengths and the limitations of our approach to detecting refactorings. We also propose new extensions to overcome the limitations.

Strengths

- **High precision and recall.** Our evaluation on the three components shows that both precision and recall of RefactoringCrawler are over 85%. Since RefactoringCrawler combines both syntactic and semantic analysis, it can process a realistic size of software with practical accuracy. Compared to other approaches [APM04, DDN00, GW05, GZ05, RD03] that use only syntactic analysis and produce large number of false positives, our tool requires little human intervention to validate the refactorings. RefactoringCrawler can significantly reduce the burden necessary to find refactoring logs that a replay tool uses to automatically upgrade component-based applications.

- **Robust.** Our tool is able to detect refactorings in the presence of noise introduced because of maintaining backwards compatibility, the noise of multiple refactorings, and the noise of renamings. Re-

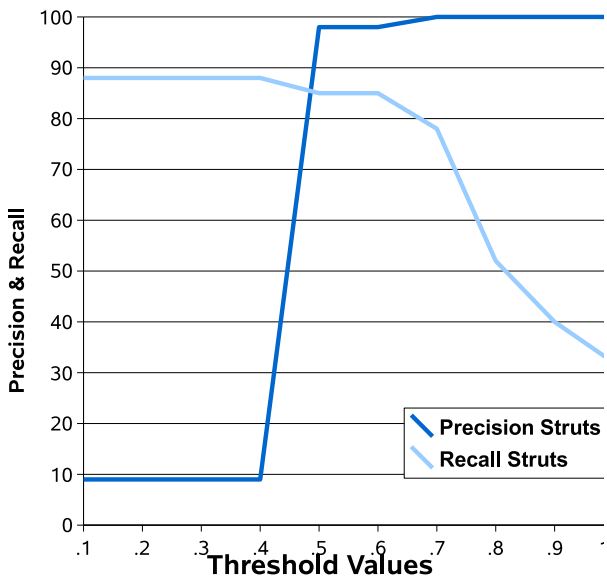

Figure 4.11: Recall and Precision vary with the value of similarity threshold for Struts case study.

namings create problems for other approaches but do not impede our tool. Since our tool identifies code entities (methods, classes, packages) based on their body resemblance and not on their names, our tool can successfully track the same entity across different versions, even when its name changes. For previous approaches, a rename is equivalent with an entity disappearing and a brand new entity appearing in the subsequent version. Another problem for previous approaches is the application of multiple refactorings to the same entity. Our tool takes this into account by sharing the log of refactorings between the detection strategies and repeating each strategy until it reaches a fixed point. Lastly, our tool detects refactorings in an open-world development where, due to backwards compatibility, obsolete entities coexist with their refactored counterparts until the former are removed. We can detect refactorings in such an environment because most of refactorings involve repartitioning the source code. This results in parts of the code from a release being spread in different places in the next release. Our algorithm starts by detecting the similarities between two versions.

- **Scalable.** Running expensive semantic analysis (like identifying similar subgraphs in the entire reference graph) on large codebases comprising of tens of thousands of nodes (methods, classes, packages) is very expensive. To avoid this, we run first a fast syntactic analysis that reduces the whole input domain to a relatively small number of candidates to be analyzed semantically. It took Refac-

toringCrawler 16 min 38 sec to analyze two versions of org.eclipse.ui containing 574 KLOC.

Limitations

- **Poor support for interfaces and fields.** Since our approach tracks the identity of methods, classes, and packages based on their textual bodies and not on their names, the lack of textual bodies can create problems. Both class fields and interface methods do not contain any body other than their declaration name. After the syntactic analysis, only entities that have a body resemblance are passed to the semantic analysis. Therefore, refactorings that happened to fields or interface methods cannot be detected. This was the case in org.eclipse.ui where between versions 2.1.3 and 3.0 many static fields were moved to other classes and many interface methods were moved to abstract classes. To counteract the lack of textual bodies for fields or interface methods, we treated their associated javadoc comments as their text bodies. This seems to work for some cases, but not all.

- **Requires experimentation.** As with any approach based on heuristics, coming up with the right values for the detection algorithms might take a few trials. Selecting threshold values too high reduces the false positives but can miss some refactorings because only those candidates that have perfect resemblance are selected. Too low threshold values produce a large number of false positives but increase the chances that all relevant refactorings are found among the results (see Fig. 4.10 and 4.11). The default threshold values for RefactoringCrawler are between 0.5 and 0.7 (for various similarity parameters) [Ref]. When default values do not produce adequate results, users could start from high threshold values and reduce them until the number of false positive becomes too large.

4.9 Summary

Syntactic analyses are too unreliable, and semantic analyses are too slow. Combining syntactic and semantic analyses can give good results. By combining Shingles encoding with traditional semantic analyses, and by iterating the analyses until a fixed point was discovered, we could detect over 85% of the refactorings in three case studies while producing less than 10% false positives.

The algorithm would work on any two versions of a system. It does not assume that the later version was produced by any particular tool. If a new version is produced by a refactoring tool that records the

refactorings that are made, then the log of refactorings will be 100% accurate. Nevertheless, there may not be the discipline or the opportunity to use a refactoring tool, and it is good to know that refactorings can be detected nearly as accurately without it.

The tool and the evaluation results are available online [Ref].

Chapter 5

Automated Replay of Refactorings

5.1 Introduction

5.1.1 Replaying Refactorings is a Special Case of Software Merging

Revisiting our upgrading theory in Section 3.3, applications are upgraded by replaying refactorings over the old version of the component and application. However, in practice there are three things that can hinder replaying. First, although the replaying views the application as a static piece of software, in reality both component and application co-evolve. Second, besides refactorings, components and applications evolve through edits. Third, the changes caused by edits and refactorings on the component as well as on the application can invalidate each other.

Figure 5.1 presents an example where refactorings and edits on the component and application side create problems for replay. Figure 5.1(i) shows an old application class `ApplicationEditor` extending a component class `AsbtractTextEditor`. In Fig. 5.1(ii), both component and application evolve. The component developers rename superclass method `performRevertOperation` → `performRevert`. In parallel, without being aware of this change, the application developer edits the subclass by adding a new method `performRevert`. In Fig. 5.1(iii), the application developer upgrades her application to use the latest version of the component. However, a naive upgrading would produce the code shown in Fig. 5.1(iii) where `AbstractTextEditor.performRevert` and `ApplicationEditor.performRevert` accidentally override each other even though the intent of the application developer was not to override the superclass method. Performing the upgrading with a refactoring replaying tool would not succeed either: although at the component site it was possible to rename the API method, at the application site this renaming results in a conflict because the application already defined a method with the new name. Currently, refactoring engines quit the operation if a such a conflict arises.

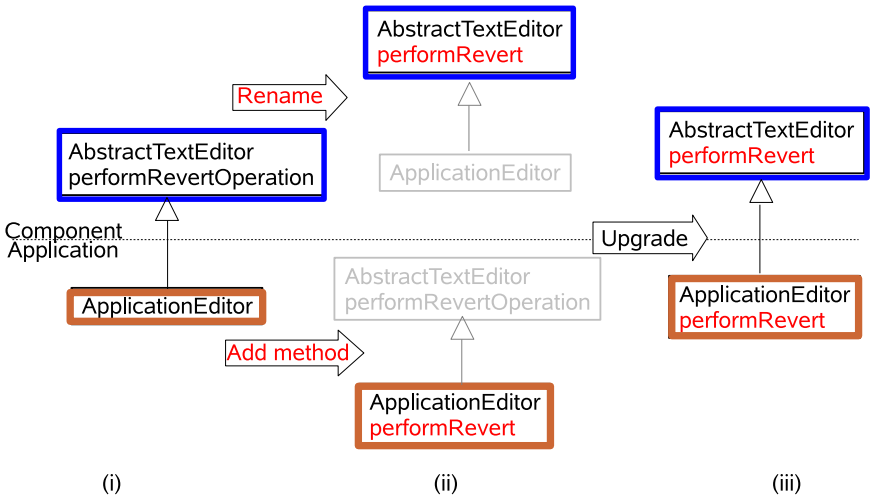

Figure 5.1: Conflicts between component and application changes.

This particular scenario resembles the more general scenario of software development: two developers check out the same version of source code, make changes in parallel, then whoever is the last to check in his changes needs to merge his changes with the previously committed changes. Revisiting the example in Fig. 5.1, one developer makes changes on the component (the classes that he touches are marked with blue), while the other developer makes changes on the application (her classes are marked with brown). Upgrading the application effectively means merging the two sets of changes. Like in software merging, the upgrading algorithm needs to signal the merge conflicts. If we can design a refactoring-aware merge algorithm that works for regular software development, the same algorithm can be used to merge refactorings and edits in components and applications.

5.1.2 The Need for a Refactoring-aware Merging Algorithm

Automated refactoring tools have become popular because they allow programmers to change source code quicker and safer than manually. However, refactoring tools make particular demands on traditional, text-based Software Configuration Management (SCM) systems.

SCM systems work best with modular systems. Different programmers tend to work on different modules and so it is easy to merge changes. But refactorings cut across module boundaries and cause changes to many parts of the system. So, SCM systems have trouble merging refactorings.

First, SCM systems signal a conflict when two programmers change the same line of code even if each just changes the name of a different function or variable. Second, text-based SCM systems are unreliable. They report merge conflicts only when two users change the same line of code. However, a merge might result in an incorrect program, even when the changes are not on the same line. This is especially true in object-oriented programs. Revisiting the example in Fig.5.1, if one user renames a virtual method while another user adds a new method in a subclass, even though these changes are not lexically near each other, textual merging could result in accidental method overriding, thus leading to unexpected runtime behavior.

This chapter describes MolhadoRef, a refactoring-aware SCM for Java, and the merge algorithm at its core. MolhadoRef has an important advantage over a traditional text-based SCM. MolhadoRef automatically resolves more conflicts (even changes to the same lines of code). Because it takes into account the semantics of refactorings, the merging is also more reliable: there are no compile errors after merging and the semantics of the two versions to be merged are preserved with respect to refactorings.

Correct merging of refactorings and manual edits is not trivial: edits can refer to old program entities as well as to newly refactored program entities. MolhadoRef uses the *operation-based* approach [LvO92]: it represents a version as a sequence of change operations (refactorings and edits) and replays them when merging. If all edits came before refactorings, it would be easy to merge the two versions by first doing a textual merge, and then replaying the refactorings. But edits and refactorings are mixed, so in order to commute an edit and a refactoring, MolhadoRef *inverts* refactorings. Moreover, refactorings will sometimes have *dependences* between them.

MolhadoRef uses Eclipse [Ecla] as the front-end for changing code and customizes Molhado [NMBT05], a framework for SCM, to store Java programs. Although the merging algorithm is independent of the Molhado infrastructure and can be reused with other SCM backends, building on top of an ID-based SCM like Molhado allows our system to keep track of the refactored entities. When evaluating MolhadoRef on a case study of three weeks of its own development and through one controlled experiment, we found that MolhadoRef merges more safely and more automatically than CVS while never losing the history of refactored entities.

MolhadoRef merges edits using the same three-way merging [MM85] of text-based SCMs. It is when MolhadoRef merges refactorings that it eliminates merge errors and unnecessary merge conflicts. So the more that refactorings are used, the more benefits MolhadoRef provides.

This chapter makes the following contributions:

- it presents an important problem that appears when merging refactored code in multi-user environments
- it presents the first algorithm to effectively merge refactorings and edit operations
- it evaluates the effectiveness of refactoring-aware merging on one real world case study and one controlled experiment

Without losing any power to merge manual edits, MolhadoRef converts refactorings from being the weakest link in an SCM system to being the strongest.

5.2 Motivating Example

To see the limitations of text-based SCM, consider the simulation of a Local Area Network (LAN) shown in Figure 5.2. This example is used as a refactoring teaching device [DRG+05] in many European universities.

Initially, there are five classes: `Packet`, a superclass `LANNode` and its subclasses `PrintServer`, `NetworkTester`, and `Workstation`. All `LANNode` objects are linked in a token ring network (via the `nextNode` variable), and they can `send` or `accept` a `Packet` object. `PrintServer` overrides `accept` to achieve specific behavior for printing the `Packet`. A `Packet` object sequentially visits every `LANNode` object in the network until it reaches its addressee.

Two users, Alice and Bob, both start from version V_0 and make changes. Alice is the first to commit her changes, thus creating version V_1 while Bob creates version V_2.

Since method `getPacketInfo` accesses only fields from class `Packet`, Alice moves method `getPacketInfo` from class `nodes.PrintServer` to `content.Packet` (τ_1). Next, she defines a new method, `sendPacket(Packet)` (τ_2), in class `NetworkTester`. The implementation of this method is empty because this method simulates a broken network that loses packets. In the same class, she also defines a test method, `testLosePacket` (τ_3) and implements it to call method `sendPacket` (τ_4). Lastly, Alice renames `WorkStation.originate(Packet)` to `generatePacket(Packet)` (τ_5). Alice finishes her coding session and commits her changes to the repository.

In parallel with Alice, Bob renames method `PrintServer.getPacketInfo(Packet)` to `getPacketInformation(Packet)` (τ_6). He also renames the polymorphic method `LANNode.send()`

Figure 5.2: Motivating example. Versions V_1 and V_2 are created in parallel and are both based on V_0. Shaded boxes show the changes in each version: τ_1 moves method `PrintServer.getPacketInfo` to `Packet`, τ_2 and τ_3 add methods, τ_4 adds a call to `sendPacket`, τ_5 renames `WorkStation.originate` to `generatePacket`, τ_6 renames `PrintServer.getPacketInfo` to `getPacketInformation`, τ_7 renames `LANNode.send` to `sendPacket`, τ_8 renames class `WorkStation` to `Workstation`.

to `sendPacket` (τ_7). Lastly, Bob renames class `WorkStation` to `Workstation` (different capitalization τ_8). Before Bob can commit his changes, he must merge his changes with Alice's.

A text-based SCM system reports merge conflicts when two users change the same line. For instance, because Alice moved the declaration of method (τ_1) while Bob altered the declaration location of the same method through renaming (τ_6), textual merging can not automatically merge these changes. This is an unnecessary merge conflict because a tool like MolhadoRef that understood the semantics of the changes can merge them.

In addition, because a text-based merging does not know anything about the syntax and semantics of the programming language, even a "successful" merge (e.g., when there are no changes to the same lines of code) can result in a merge error. Sometimes errors can be detected at compile-time. For instance, after textual merging, the code in method `testLosePacket` does not compile because it calls method `send` whose declaration was replaced by `sendPacket` through a rename (τ_7). Such an error is easy to catch, though it is annoying to fix.

Other errors result in programs that compile but have unintended changes to their behavior. For instance, because Alice introduces a new method `sendPacket` in subclass `NetworkTester` and Bob renames the polymorphic method `send` to `sendPacket`, a textual merge results in accidental method overriding. Therefore, the call inside `testSendToSelf` to `sendPacket` uses the empty implementation provided by Alice(τ_2) to simulate loss of packets, while originally this method call used the implementation of `LANNode.send`. Since this type of conflict is not reported during merging or compilation, the user can erroneously assume that the merged program is correct, when in fact the merging introduced an unintended change of behavior.

Figure 5.3 shows the merged program after merging with MolhadoRef. MolhadoRef catches the accidental method overriding caused by adding a new method (τ_2) and renaming another method (τ_7) and presents it to the user. The user decides to rename the newly introduced method `NetworkTester.sendPacket` to `losePacket`. This is the only time when MolhadoRef asks for user intervention; it merges automatically all the remaining changes. The merged version contains all the edits and refactorings performed by Alice and Bob (e.g., notice that method `getPacketInformation` is both renamed and moved).

THE MERGED VERSION

PrintServer.java

```
...
public class PrintServer extends LANNode {
   public void print(Packet p) {
      String packetInfo =
        p.getPacketInformation();
      System.out.println(packetInfo);
   }
}
```

Packet.java

```
...
public class Packet {
  ...
  public String getPacketInformation ()
  {
    String packetInfo = originator + ": " +
       addressee + "[" + contents + "]";
    return packetInfo;
  }
}
```

LANNode.java

```
...
public class LANNode {
   ...
   protected void sendPacket(Packet p) {
      System.out.println(name + nextNode.name);
      this.nextNode.accept(p);
   }
}
```

NetworkTester.java

```
...
public class NetworkTester extends LANNode {
  public void testSendToSelf() {
     Packet packet = new Packet ();
     packet.addressee = this;
     packet.originator = this;
     sendPacket (packet);
  }
  public void accept (Packet p) {
     ...
  }
  protected void losePacket(Packet p){
  } // left empty to drop packets
  void testLosePacket(boolean losePacket) {
    Packet packet = new Packet ();
    packet.addressee = new LANNode ();
    packet.originator = this;
    if (losePacket) losePacket(packet);
    else sendPacket(packet);
  }
}
```

Workstation.java

```
...
public class Workstation extends LANNode {
  ...
}
```

Figure 5.3: Resolved motivating example using MolhadoRef.

5.3 Background and Terminology

Our approach to refactorings-tolerant SCM systems is based on a different paradigm, called *operation-based merging* [LvO92]. In the operation-based approach, an SCM tool records the operations that were performed to transform one version into another and replays them when updating to that version. An operation-based system treats a version as the sequence of operations used to create it.

Our goal is to provide merging at the API level, that is, our merging algorithm aims for a correct usage of all the APIs. For this reason, we distinguish between operations that affect the APIs and those that do not. MolhadoRef treats a version as composed of the following three operations: API refactorings, API edits, and code edits. MolhadoRef handles the following API refactorings: rename package, rename class, rename method, move class, move method, and changing the method signature (these were among the most popular refactorings found in previous studies [DJ06]). MolhadoRef handles the following *API edits*: added package, deleted package, added class, deleted class, added method declaration, deleted method declaration, added field declaration, deleted field declaration. Any other types of edits are categorized as *code edits*.

Code edits do not have well defined semantics, making it difficult to merge them correctly. API edits have better defined semantics. But refactorings are the operations with the most well defined semantics, so

the ones that can benefit the most from operation-based merging. Therefore, MolhadoRef merges code edits textually and since it is aware of the semantics of refactorings and API edits, it merges them semantically.

We regard the program changes as source-to-source program transformations. When necessary, we make the distinction between refactorings (ρ) and edits (σ). Refactorings are transformations that preserve the semantics, while edits usually change the semantics of programs.

We remind the reader about some of the terminology introduced in Section 3.1. Operations usually have preconditions, and they can be composed (denoted by ";"). Operations can *commute*, *conflict*, or be *dependent* (denoted by $\prec_P$) upon other operations. For example, τ_2 and τ_8 from our motivating example commute. Definition 3 describes conflicts that produce compile errors. MolhadoRef also catches some conflicts that produce run-time errors. MolhadoRef currently catches conflicts that involve method overriding, such as the accidental method overriding between τ_2 and τ_7.

An example of dependence is the renaming of method `WorkStation.originate` to `generatePacket` done by Alice (τ_5) and the renaming of class `WorkStation` to `Workstation` done by Bob (τ_8). If τ_8 is played first, the replaying of τ_5 is not possible because at this time the fully qualified name `WorkStation.originate` no longer exists, thus $\tau_5 \prec_P \tau_8$.

This dependence between τ_5 and τ_8 exists because current refactoring engines are based on the names of the program entities, and class `WorkStation` no longer exists after replaying τ_8. If the refactoring engine used the IDs of the program elements, changing the names would never pose a problem [DNJ06]. To make name-based refactoring engines be ID-based requires rewriting the engine. This is unfeasible, so the next best solution is to *emulate* ID-based engines.

To make the current name-based refactoring engines emulate ID-based ones, there are at least two approaches. The first is to reorder the refactorings (e.g., rename method `WorkStation.originate()` before rename class `WorkStation`). The second is to modify the refactoring engine so that when it changes source code, it also changes subsequent refactorings. For example, during the replay of renaming class `WorkStation` to `Workstation`, the refactoring engine changes the subsequent refactoring `RenameMethod(WorkStation.originate, generatePacket)` to `RenameMethod(Workstation.originate, generatePacket)`. Our merging algorithm uses both approaches.

Consider a scenario where Alice renames method `m1` in superclass `A` (call it operation ρ_1), and Bob

desires to override `A.m1` by adding a method `m1` in subclass `B` (call it operation σ_1). Applying these two operations in either order produces a valid program. However, only one order preserves Bob's intent: applying the edit followed by renaming the method in the superclass preserves the overriding relationship since the renaming ρ_1 also updates the edit σ_1 (renaming a method updates all overriding methods in a class hierarchy). The other order, the renaming ρ_1 followed by the edit σ_1 would result in a program that compiles, but `B.m1` no longer overrides the superclass method, violating Bob's intent. Thus, there is a dependence $\sigma_1 \prec_P \rho_1$.

5.4 Merging Algorithm

5.4.1 High Level Overview

We illustrate the merging algorithm (see pseudocode in Fig. 5.4) using the LAN simulation example presented earlier. The details of each module are found in the later subsections.

The merging algorithm takes as input three versions of the software: version V_0 is the base version and V_1 and V_2 are derived from V_0. In addition, the algorithm takes as input the refactorings that were performed in V_1 and in V_2. These refactoring logs are recorded by Eclipse's refactoring engine. The output is the merged version, V_merged, that contains edits and refactorings performed in V_1 and V_2.

Step #1 detects the changes that happened in V_1 and V_2 by performing a 3-way comparison between V_1, V_2 and V_0. From these changes and the refactoring logs, it categorizes edits and refactorings operations. For example, in V_1 it detects two added methods, τ_2 and τ_3. In V_2 it detects no edits but only refactorings.

Step #2 searches for compile and run-time conflicts in API edits and refactorings. In our example it detects a conflict between the add of a new method, τ_2 in V_1, and the rename method refactoring, τ_7 in V_2. This conflict reflects an accidental method overriding. The conflict is presented to the user who resolves it by choosing a different name for the added method (in this case `losePacket` instead of `sendPacket`). The algorithm also searches for possible circular dependences between operations performed in V_1 and operations in V_2. If any are found, the user deletes one of the operations involved in the cycle (in our example there are no circular dependences). This process of detecting/solving continues until no more conflicts or circular dependences remain.

Step #3 inverts each refactoring in V_1 and V_2 by applying another refactoring. For instance, it inverts the

```
1
2  INPUT = {V_2, V_1, V_0, refactoringLogs}
3
4  //Step #1 Detecting Operations
5  Operations ops= 3-wayComparison(V_2,V_1,V_0)
6  Operations refactorings= detectRefactorings(refactoringLogs)
7  Operations edits= detectEdits(ops, refactorings)
8
9  //Step #2 Detecting and Solving Conflicts and Circular Dependences
10 repeat{
11   Conflicts conflicts = detectConflicts(edits, refactorings)
12   {edits, refactorings}= userSolvesConflicts({edits, refactorings}, conflicts)
13   Graph operationsGraph = createDependenceGraph(refactorings, edits)
14   {refactorings, edits, operationsGraph} = userRemovesCircularDependences(operationsGraph)
15 } until noConflictsOrCircularDependences(refactorings, edits, operationsGraph)
16
17 //Step #3 Inverting Refactorings
18 Version V_1_minusRefactorings= invertRefactorings(V_1, refactorings)
19 Version V_2_minusRefactorings= invertRefactorings(V_2, refactorings)
20
21 //Step #4 Textual Merging
22 Version V_merged_minusRefactorings= 3-wayTextualMerge(V_2_minusRefactorings,
23                                                       V_1_minusRefactorings,V_0)
24
25 //Step #5 Replaying Refactorings
26 Operations orderedRefactorings= refactoringsTopologicalSort(operationsGraph)
27 Version V_merged= replayRefactorings(V_merged_minusRefactorings, orderedRefactorings)
28
29 OUTPUT = {V_merged}
```

Figure 5.4: Overview of the merging algorithm.

move method refactoring τ_1 by moving method `getPacketInfo` back to `PrintServer`, and it inverts the rename class refactoring τ_8 by renaming `Workstation` back to `WorkStation`. By inverting refactorings, all the edits that were referencing the refactored program entities are still kept in place, but changed to refer to the old version of these entities. This step produces two software components, $V_1^{-Refactorings}$ and $V_2^{-Refactorings}$, that contain all the changes in V_1, respectively V_2, except refactorings.

Step #4 merges textually (using the classic three-way merging [MM85]) all the API and code edits from $V_1^{-Refactorings}$ and $V_2^{-Refactorings}$. Since the refactorings were previously inverted, all same-line conflicts that would have been caused by refactorings are eliminated. For example, inside `PrintServer.print` there are no more same-line conflicts. Therefore, textual merging of code edits can proceed smoothly. This step produces a software component, called $V_{merged}^{-Refactorings}$.

Step #5 replays on $V_{merged}^{-Refactorings}$ the refactorings that happened in V_1 and V_2. Before replaying, the algorithm reorders all the refactorings using the dependence relations. Replaying the refactorings incor-

porates their changes into $V_{merged}^{-Refactorings}$, which already contains all the edits. For example, replaying a method renaming refactoring updates all the call sites to that method, including the ones introduced by edits.

5.4.2 Detecting Operations

To detect refactorings, API edits and code edits, the algorithm takes as input the source code of the three versions V_0, V_1, V_2, along with their refactoring logs. Recent extensions to refactoring engines (e.g., [Eclb]) log the refactorings at the time they were performed. This log of refactorings is saved in a configuration file and is stored along with the source code. Since our algorithm is implemented as an Eclipse plugin, it has access to this log of refactorings. Even in cases when such a log of refactorings is not recorded, it can be detected using RefactoringCrawler [DCMJ06], a tool for inferring refactorings.

To detect the API edits and code edits, the algorithm employs a three-way textual *comparison* (a two-way comparison cannot distinguish between additions and deletions [Men02]). This comparison (line 5 in Fig. 5.4) detects lines, files, and folders that were changed. From this low level information, the algorithm constructs (line 7) the higher level, semantic API edits (e.g., add method) by parsing and correlating the positions of changed tokens with that of changed lines.

Even though the scope of our merging is at the API level, to correctly signal compile- or run-time conflicts, the algorithm detects a few edit operations that are below the API level. These include add/delete method call, add/delete class instantiation, add/delete class inheritance, add/delete typecast. From this information, the algorithm is able to detect conflicts like the one appearing if Alice deletes the method declaration `accept` and Bob adds a method call to `accept`.

Some of the edit operations overlap with or are the side effects of refactorings. For example, after renaming class `WorkStation` to `Workstation`, a textual comparison renders `WorkStation` as deleted and `Workstation` as added. Using the information from the refactoring logs, the algorithm discards these two edits since they are superseded by the higher level refactoring operation.

The output of this step is the list of change operations (refactorings and edits) that happened in each of V_1 and V_2.

5.4.3 Detecting and Solving Conflicts and Circular Dependences

Detecting Conflicts. Next, MolhadoRef detects conflicts between operations in V_1 and V_2. For this, it uses a *conflict matrix* (the companion tech report [DMJN06] describes all its cells). For any two kinds of operations, the matrix gives a predicate that responds whether the operations conflict. This matrix includes refactorings, API edits, and the code edits that are currently handled. MolhadoRef instantiates the conflict matrix for the concrete operations detected in the previous step and signals any conflicts between these operations.

Next we present the content of one single cell in the conflict matrix, namely the case when τ_i is RenameMethod(m_1, m_2) and τ_j is RenameMethod(m_3, m_4). These two renamings result in a conflict if (i) the source of both refactorings is the same program element (e.g., m_1 = m_3) but their new names would be different, or (ii) the source of both refactorings are different program elements but the destination of both refactorings is the same program element (e.g., $m_2 = m_4$). In addition, due to polymorphic overriding, we must also consider the case when two methods are not the same program element, but one method overrides the other.

When the source of both refactorings are the same (i), if methods m_1 and m_3 are in the same class, there would be a compile-time conflict since the users want to rename the same method differently. If the methods m_1 and m_3 are overriding each other, renaming them differently results in a run-time conflict because the initial overriding relationship would be broken. When the destination of the two refactorings is the same (ii), if methods m_1 and m_3 are in the same class, renaming them to the same name results in a compile-time error (two methods having the same signature and name). If methods m_1 and m_3 are not in the same class and do not initially override each other, renaming them to the same name results in a run-time conflict because of accidental method overriding.

More formally, using first-order predicate logic, Fig. 5.5 describes the content of the RenameMethod / RenameMethod cell in the conflict matrix. Similar formulae describing the remaining cells in the matrix are in a companion tech report [DMJN06]. The predicates in each cell are computed "by hand", but they are carefully revised.

Although the matrix and its predicates describe operations independent of a particular program, in this step MolhadoRef instantiates it for the program under analysis and its concrete operations detected in step #1. For example, MolhadoRef detects the accidental method overriding conflict between τ_2 and τ_7 in the

$$hasConflicts(RenameMethod(m_1, m_2), RenameMethod(m_3, m_4)) :=$$
$$((m_1 = m_3 \vee overrides(m_1, m_3)) \wedge (simpleName(m_2) \neq simpleName(m_4)))$$
$$\vee$$
$$((m_1 \neq m_3 \wedge \neg overrides(m_1, m_3)) \wedge (m_2 = m_4 \vee overrides(m_2, m_4)))$$

Figure 5.5: The RenameMethod / RenameMethod cell in the conflict matrix.

motivating example. This conflict is presented to the user who can decide how to solve it.

Detecting Circular Dependences. In this step, the merge algorithm also creates the dependence graph (line 13 in Fig. 5.4) between operations performed in the two versions to be merged. Initially, there is a total (i.e., linear) order of the change operations in each version, given by the time sequence in which these operations were applied. However, when merging, the operations can be replayed in any order, unless there is a dependence between them. Thus, the total order can be ignored in favor of a partial order, induced by the $\prec_P$ relation.

To create this partial order, we represent each operation as a node in a directed graph, called the *dependence graph*. When $\tau_i \prec_P \tau_j$, the algorithm adds a directed edge from τ_i to τ_j. To find out the $\prec_P$ dependences, the algorithm uses a *dependence matrix* which describes dependence relations between all kinds of operations (similar to how the *conflict matrix* describes conflicts). MolhadoRef instantiates the dependence matrix for the concrete operations in the versions to be merged. MolhadoRef places all the concrete operations in the dependence graph and adds dependence edges using the information from the dependence matrix.

Next, the algorithm searches for cyclic dependences in the dependence graph. There can only be cycles between operations from different users, not between operations from the same user because for each user it was initially possible to play all the operations. Figure 5.6 shows a scenario where a cycle appears between operations from two users. Initially, the base version contains one class `A` with one method `A.m1`. The operations on the left hand side are performed by Bob, the one on the right hand side by Alice. Bob renames class `A` to `B`, then adds a subclass `C` of `B`. Next, in class `C` Bob adds a method with name `m1` which overrides the method in the superclass `B`. In parallel with Bob, Alice renames method `A.m1` to `m2`.

The arrows on Bob's side indicate the original order in which the operations took place. The arrow from Alice's rename method to Bob's renaming the class, points to a dependence caused by the current refactoring

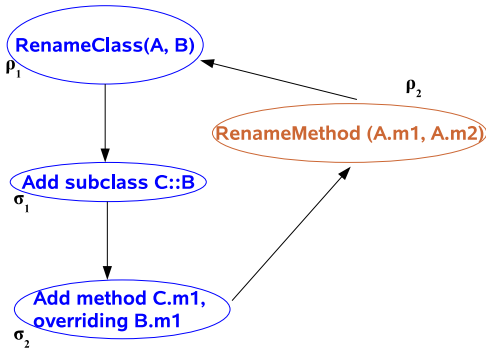

Figure 5.6: Circular dependence between operations from two users. Arrows represent the direction of the dependences (e.g., add method C.m1 must be applied before renaming method A.m1). Left hand side shows operations from Bob (with blue color), right hand side shows operations from Alice (with orange color).

engines. The refactoring engines use the fully qualified names to identify the program elements to be refactored, therefore, renaming the method `A.m1` must be performed before renaming its class declaration `A`, otherwise the refactoring engine can no longer find the element `A.m1`. The arrow from Bob's adding method `C.m1` to Alice's renaming the superclass method A.m1 points to another dependence. The subclass method must be added before the renaming of the superclass method `A.m1` such that when replaying the `RenameMethod(A.m1, m2)`, it also renames `C.m1` (playing them in a different order would cause the two methods to no longer override each other).

After it finds all cycles, MolhadoRef presents them to the user who must choose how to eliminate cycles (read next subsection). Assuming that there are no more cycles, all operations are in a directed acyclic graph.

User-assisted Conflict and Dependence Resolution. Circular dependences and compile and run-time conflicts require user intervention. To break circular dependences, the user must select operations to be discarded and removed from the sequence of operations that are replayed during merging. Discarding refactorings has no effect on the semantics of the merged program because refactorings are transformations that do not change the semantics. Discarding edits can potentially affect the semantics of the merged program. However, this solution would be used only in extreme cases (we have never run into such a scenario during evaluation). Alternatively, most circular dependences (including the one in Fig. 5.6) can be solved automatically by MolhadoRef by inverting the refactorings (see section 5.4.4).

To solve the syntactic or semantic conflicts caused by name collision, the user must select a different

name for one of the program elements involved in the conflict. In our LAN motivating example (Fig. 5.2), Alice renames method `send` to `sendPacket` and Bob adds a new method declaration `sendPacket` such that the two methods accidentally override each other. This conflict is brought into Bob's attention who can either choose a different name for his newly introduced method, or can pick a new name to supersede the name chosen by Alice. In the motivation example, Bob chose to rename his newly introduced method `sendPacket` to `losePacket`. Once Bob chose the new name, MolhadoRef automatically performs this rename.

The process of finding and solving conflicts and circular dependences is repeated until there are no more conflicts or circular dependences (line 15 in Fig.5.4). The algorithm always converges to a fixed point because it starts with a finite number of operations and user deletes some in each iteration.

5.4.4 Inverting Refactorings

Step #3 makes a version of V_1 and V_2 without any refactorings by inverting all refactorings. Inverting a refactoring ρ_1 involves creating and applying an *inverse* refactoring. ρ_1^- is an inverse of ρ_1 if $\rho_1^-(\rho_1(P))$ = P for all programs P that meet the precondition of ρ_1. For example, the inverse of the refactoring that renames class `A` to `B` is another refactoring that renames `B` to `A`, the inverse of a move method is another move method that restores the original method, the inverse of extract method is inline method, the inverse of pull-up member in a class hierarchy is push-down member. In fact, many of the refactorings described in Fowler's refactoring catalog [FBB+99] come in pairs: a refactoring along with its opposite (inverse) refactoring.

Given any refactoring, there exists another refactoring that inverts (undoes) the first refactoring (although such inverse refactoring cannot be always applied because of later edits). There is an important distinction between what we mean by inverting a refactoring and how the popular refactoring engines (like Smalltalk RefactoringBrowser, Eclipse or IntellijIdea) undo a refactoring. To decrease memory usage and avoid re-computations of preconditions, the refactoring engines save the location of all source code that was changed by the refactoring. When undoing a refactoring, the engines undo the source changes of these locations.

Although efficient, this approach has a drawback: the only way to undo a refactoring that was followed by edits is to first undo all the edits that come after it. This approach is not suitable for MolhadoRef. MolhadoRef must be able to undo a refactoring without undoing later operations. Thus MolhadoRef inverts

refactorings by creating and executing an inverse refactoring operation (which is another refactoring).

To create the inverse refactoring, MolhadoRef uses the information provided in the refactoring logs of Eclipse. Each refactoring recorded by Eclipse is represented by a refactoring descriptor which contains enough textual information to be able to recreate and replay a refactoring. Among others, the descriptor contains information such as what kind of refactoring is created, what is the program element on which it operates, what other parameters have been provided by the user through the UI (e.g., the new name in case of a rename refactoring or the default value of an argument in case of refactoring that changes a method signature by adding an argument). Out of this information, MolhadoRef creates another refactoring descriptor, which represents the inverse refactoring. For example, the inverse of the move method refactoring descriptor representing τ_1 in our LAN example, is another move method descriptor representing a refactoring that moves method `Packet.getPacketInfo` back to `PrintServer`.

From the inverse refactoring descriptor, MolhadoRef creates and initializes an Eclipse refactoring object. Once the refactoring object is properly initialized, the refactoring is executed using Eclipse's refactoring engine.

Inverting a refactoring and executing the inverse refactoring also changes the edits. Recall the motivation example where Bob renames method `getPacketInfo` to `getPacketInformation`. Later he adds a new method call to `getPacketInformation`. By inverting the rename method refactoring with the inverse refactoring (renaming `getPacketInformation` to `getPacketInfo`), the new call site to `getPacketInformation` is updated too, while keeping the call site in the same place. Deleting the call site altogether would have introduced a different behavior, while leaving the call site untouched would have produced a compilation error.

Probably the most notable aspect of inverting refactorings is that it inverts the dependences between edits and refactorings, allowing refactorings to come after edits, thus it changes a $refactoring \prec_P edit$ dependence into $edit \prec_P refactoring$ dependence. This has two advantages. First, when replaying refactorings in step #5, the fact that refactorings come after edits ensures that all changes caused by refactorings are incorporated into edits. Second, inverting refactorings automatically breaks most of the circular dependences between operations. Recalling the example from Fig. 5.6 with circular dependence, Fig. 5.7 shows the dependence graph after inverting the rename class refactoring. Notice that the edits refer now to class `A` instead of `B`, and there are no more circular dependences.

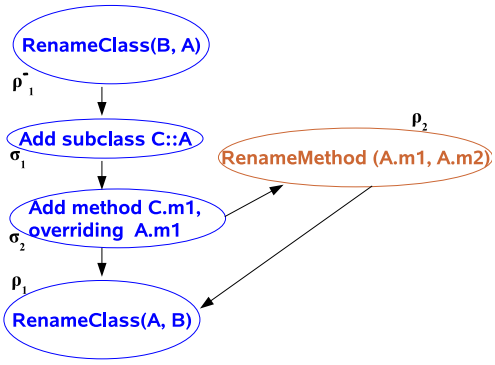

Figure 5.7: Resolved example of circular dependence from Fig. 5.6. Inverting the rename class refactoring ρ_1 by applying the inverse refactoring (ρ_1^-) breaks the circular dependence.

Just as refactorings have preconditions, inverting a refactoring has preconditions too, and if those preconditions are not met, then a refactoring cannot be inverted. Edits in V_1 and V_2 that were applied after refactorings could break the preconditions of inverse refactorings. To handle such cases, we have three heuristics: adding program transformations, storing additional information before inverting a refactoring, or a fall-back heuristic in case that others fails.

Heuristic #1: Renaming a program element. This heuristic renames a program element to a unique name when name collisions prevent inverting a refactoring. For example, if Bob renames `PrintServer.getPacketInfo` to `getPacketInformation` (τ_6) and later adds a new method `getPacketInfo` in the same class, inverting the rename refactoring in step #3 is not possible because the name `getPacketInfo` is already in use (by the lately added method). MolhadoRef searches for potential name collisions before inverting the refactoring, and executes another renaming to avoid the collision. In this case, before inverting the refactoring, the algorithm renames the newly introduced method `getPacketInfo` to a unique name, say `getPacketInfoABC123` and tags this rename refactoring. Now that there are no more name collisions, renaming τ_6 can be inverted. Later, in step #5, after all the regular refactorings have been replayed, the algorithm inverts all refactorings marked with tags. Thus, it renames the new method `getPacketInfoABC123` back to `getPacketInfo`. At this stage, there are no more name collisions because τ_6 would have executed.

Heuristic #2: Storing additional information. This heuristic stores additional information before inverting a refactoring, since some information can get lost when inverting refactorings. Consider the case when Bob changes the signature of a method `sendPacket` by adding an extra argument of type integer

with the default value 0 to be used in method calls. Later he adds a call site where he passes the value 7 for the extra argument. Inverting the refactoring and replaying it naively would lose the value 7 and replace it with value 0. Before inverting the refactoring, MolhadoRef saves the location of the new call sites and the values of parameters so that it can restore the same values later when replaying the refactoring in step #5.

Heuristic #3: Fall-back - treating refactorings as edits. When no heuristic for inverting a refactoring is found, the algorithm treats the refactoring as a classic textual edit, namely, the refactoring is not inverted and replayed, but its code changes are incorporated by textual merging. Although the advantages of incorporating the semantics of the refactoring are lost, the algorithm can always make progress and in the worst case it is as good as classic textual merging.

The first two heuristics are sufficient to invert all the refactorings in the evaluations we have done. Nevertheless, further evaluations might require developing new heuristics to handle other types of refactorings, or force us to use the fall-back heuristic (heuristic #3). An analysis of all refactorings currently supported in Eclipse shows that all these refactorings could be inverted by first renaming conflicting program elements (heuristic #1) or storing additional information before inverting the refactoring (heuristic #2).

5.4.5 Textual Merging

Once refactorings are inverted, all the edits in V_1 and V_2 that referred to the refactored APIs now refer to the APIs present in version V_0. The algorithm merges textually all files that were changed by edits using the three-way merging [MM85] that most text-based SCMs use.

All code changes inserted by refactorings that would have caused same-line or same-block conflicts are eliminated due to the fact that refactorings were previously inverted. In our LAN example, although both users changed the declaration of `getPacketInfo` (τ_1 and τ_6), after inverting the refactorings, the call to method `getPacketInfo` inside `PrintServer.print` no longer causes same-line conflict.

Still, if two users change the same lines by code edits (not refactorings), this can generate a same-line conflict requiring user intervention, although MolhadoRef can automatically merge a few more edits than textual-based merging. For example, if Alice and Bob each add a new method declaration at the same position in a source file, MolhadoRef merges this automatically using the semantics of API edits. In contrast, textual-based merging would signal a same-line conflict. However, it is when multiple refactorings affect the same lines that MolhadoRef shines over text-based merging.

5.4.6 Replaying Refactorings

Current refactoring engines identify program entities with fully qualified names. Within a stream of operations from a single version, names will always be consistent because each refactoring works with the current names of program elements. But when refactorings are merged from two different streams, renamings can interfere with each other in two ways.

The first is where the refactorings refer to two different entities, but one of them has a name that includes the other. For example, the fully qualified name of a method includes the name of its class. If one refactoring renames a class, and the other changes a method in that class, it is important that the right method gets changed. MolhadoRef solves this problem by making sure that the refactorings of a method are performed before the refactorings that rename its class. More precisely, MolhadoRef uses a topological sort algorithm [CLRS01] to reorder the nodes in the refactorings DAG created in step #2.

The second is where two refactorings refer to the same entity. Sometimes this is a conflict that must be resolved by the user, such as when the two refactorings change the name of the same entity. This case would have been resolved by step #2. So, the only remaining cases are when the two refactorings change the same entity, but in different ways. For example, one refactoring could rename a method, and the other could move it to a new class (e.g., τ_6 and τ_1). Changing either the method name or the class name will invalidate the other refactoring. Mohaldoref solves this problem by modifying refactorings. If a refactoring from one version is replayed after a rename or a ”move method” refactoring from the other version, second refactoring is changed to use the new name. This lets a name-based system like Eclipse emulate an ID based system.

To handle multiple refactorings to the same element, we extended the definition and semantics of a refactoring. In addition to source code, a refactoring changes subsequent refactorings in a chain of refactorings. An *enhanced* refactoring is a transformation from source code and a chain of refactorings to another source code and another chain of refactorings. Conceptually, our enhanced refactoring, $\rho^{Enhanced}$ is the pair of a classic refactoring transformation, ρ, with another transformation, θ, that changes subsequent refactorings in a chain:

$\theta : Refactorings \rightarrow Refactorings$

$\rho^{Enhanced} =< \rho, \theta >$

Composing an enhanced refactoring with another refactoring changes the second refactoring:

$\rho_i^{Enhanced}; \rho_j = < \rho_i, \theta_i >; \rho_j = \rho_i; (\theta_i(\rho_j))$

Each θ transformation is dependent upon the type of enhanced refactoring from which it is a part. For instance, a θ_{Ren} transformation applied on a move refactoring changes the parameters of the move refactoring:

$$\theta_{Ren(m \rightarrow k)}(\rho_i) = \begin{cases} Mov(k \rightarrow p) & \text{if } \rho_i = Mov(m \rightarrow p) \\ Mov(z \rightarrow p) & \text{if } \rho_i = Mov(z \rightarrow p) \end{cases} \tag{5.1}$$

Applying θ_{Ren} on an empty chain of refactorings is equivalent to applying an identity function:

$\rho_{Ren}^{Enhanced}([]) = \rho_{Ren}; (\theta_{Ren}([])) = \rho_{Ren}$

Given a chain $C = [\rho_i, \rho_{i+1}, ..., \rho_k]$, applying a θ transformation on the whole chain C incorporates the effect of the renaming into the whole chain:

$\theta_{Ren}([\rho_i, \rho_{i+1}, ..., \rho_k]) =$

$(\theta_{Ren}(\rho_i)); (\theta_{Ren}(\rho_{i+1})); ...; (\theta_{Ren}(\rho_k))$

The presence of θ transformations elegantly solves cases when multiple refactorings affect the same program element. Figure 5.8 presents the composition of two enhanced refactorings, a rename and a move method, that change the same program element, `PrintServer.getPacketInfo`. Each enhanced refactoring is decomposed into the classic refactoring and its θ transformation. The enhanced rename method refactoring changes the arguments of the subsequent move method so that the move method refactoring operates upon element `PrintServer.getPacketInformation`.

5.5 Discussion

Our approach relies on the existence of logs of refactoring operations. However, logs are not always available. To exploit the full potential of record & replay of refactorings, we developed RefactoringCrawler [DCMJ06] to automatically detect the refactorings used to create a new version. These inferred refactorings can be fed into MolhadoRef when recorded refactorings are not available.

$$\rho^{Enhanced}_{Ren(getPacketInfo \rightarrow getPacketInformation)}; \rho^{Enhanced}_{Mov(PrintServer.getPacketInfo \rightarrow Packet.getPacketInfo)} =$$
$$= \rho_{Ren(getPacketInfo \rightarrow getPacketInformation)}; (\theta_{Ren}(\rho^{Enhanced}_{Mov(PrintServer.getPacketInfo \rightarrow Packet.getPacketInfo)}))$$
$$= \rho_{Ren(getPacketInfo \rightarrow getPacketInformation)}; \rho^{Enhanced}_{Mov(PrintServer.getPacketInformation \rightarrow Packet.getPacketInformation)}$$
$$= \rho_{Ren(getPacketInfo \rightarrow getPacketInformation)}; (\rho_{Mov(PrintServer.getPacketInformation \rightarrow Packet.getPacketInformation)}; (\theta_{Mov}([])))$$
$$= \rho_{Ren(getPacketInfo \rightarrow getPacketInformation)}; \rho_{Mov(PrintServer.getPacketInformation \rightarrow Packet.getPacketInformation)}$$

Figure 5.8: Composition of enhanced refactorings: one refactoring renames `PrintServer.getPacketInfo`, the other refactoring moves this method to class `Packet`.

Although one might expect that circular dependences would require a lot of manual editing, in practice such dependences are rare. Circular dependences can be eliminated manually by deleting some of the operations in the cycle, or automatically by inverting refactorings (as seen in Fig. 5.7) or by the *enhanced* refactorings (step #5). The enhanced refactorings eliminate dependences between refactorings that change the same program elements since these refactorings can be replayed in any order.

Our merging algorithm discriminates between refactorings and API edits. Although both of these operations have semantics that can be easily inferred by tools, MolhadoRef inverts and replays refactorings only, not API edits (although API edits' semantics are taken into account during conflict detection). Conceptually, API edits can be treated the same way how MolhadoRef currently treats refactorings. There are two reasons why MolhadoRef treats them differently. First, API edits are harder to invert than refactorings since API edits are not behavior-preserving. Inverting an `AddAPIMethod` by removing the method would invalidate all the edits that refer to the new method (e.g., method call sites). To fix this, it would require inverting also the code edits (e.g., removing the call sites that refer to new method). Second, API edits do not have the same global effect as the API refactorings, because only one user (e.g., the one that introduced the new method) would be aware of the new API, leading to fewer cases of same-line conflicts than when refactorings are involved. Since there are far less benefits from inverting and replaying API edits, we decided to treat them like code edits.

MolhadoRef is built on top of the Molhado object-oriented SCM infrastructure [NMBT05], which was developed for creating SCM tools. Molhado is a database that keeps track of history. MolhadoRef translates Java source code (all Java 1.4 syntax is supported) into Molhado structures. At the time of check-in, MolhadoRef parses to the level of method and field declaration and creates a Molhado counterpart for each

program element. The method/field bodies are stored as attributes of the corresponding declarations. For each entity, Molhado gives a unique identifier. When refactorings change different properties of the entities (e.g., names, method arguments), MolhadoRef updates the corresponding Molhado entries. Nevertheless, the identity of program entities remains intact even after refactoring operations (for a detailed description on implementation, see [DMJN06] and [DNJ06]).

Can such a refactoring-aware SCM system be implemented on top of a traditional SCM that lacks unique identifiers? We believe that with enough engineering, the features of MolhadoRef can be retrofitted on a system like CVS. To retrieve history of refactored elements, it is important to keep a record of unique identifiers associated with program elements. Identifier-to-name maps can be saved in metadata files and stored in the repository along with the other artifacts. At each check-out operation, the MolhadoRef CVS client needs to load these metadata files into memory so that they can be updated as the result of refactoring operations. At each check-in operation, these files are stored back into the repository.

Limitations of MolhadoRef: One obvious limitation is that our approach requires that the SCM be language-specific. However, we do not see this as a limitation, but as a trade-off: we are intentionally giving up generality for gaining more power. This is no different than other tools used in software engineering, for example, IDEs are language specific (along with all the tools that make-up an IDE, e.g., compilers, debuggers, etc.), refactoring tools are all language-specific. Second, adding support in MolhadoRef for a new kind of refactoring entails adding several cells in the conflict and dependence matrices, describing all combinations between the new operation and all existing operations. This can be a time-consuming task. We discovered that new cells tend to reuse predicates from earlier cells, which makes them easy to implement. Cells requiring new predicates are still time consuming to implement. Third, the correctness of the system depends on the correctness of formulae in the conflict and dependence matrices. However, we carefully revisited those formulae. In addition, multiple experiments and more experience using our system can help to empirically test the correctness of those formulae.

5.6 Controlled Experiment

We want to evaluate the effectiveness of MolhadoRef in merging compared to the well known text-based CVS. For this, we need to analyze source code developed in parallel that contains both edits and refactorings. Software developers know about the gap between existing SCM repositories and refactorings tools. Since

developers know what to avoid, notes asking others to check in before refactorings are performed, are quite common. Therefore, it is unlikely that we will find such data in source code repositories. As a consequence, we designed a controlled experiment.

5.6.1 Hypotheses

Operation-based merging has intuitive advantages over text-based merging since it is aware of the semantics of change operations. We hypothesize that operation-based merging is more effective than text-based merging. Namely, MolhadoRef:

- **H1** automatically solves more merge conflicts
- **H2** produces code with fewer compile-time errors
- **H3** produces code with fewer run-time errors
- **H4** requires less time to merge

than CVS.

5.6.2 Experiment's Design

We wanted to recreate an environment, similar to the regular program maintenance, where developers add new features, run regression test suites and make changes in the code base in parallel. However, we wanted an environment where software developers did not know and worry about other people working on the same code base. We randomly split 10 software developers into two groups, G1 and G2, each group containing 5 developers. Each developer in group G1 was asked to implement feature ACK, while each developer in group G2 implemented feature MAX_HOPS. All developers were given the same starting code base. At the end, we took their code, stored it in both CVS and MolhadoRef and merged their changes using Eclipse's CVS client and MolhadoRef.

Demographics We asked 10 graduate students at UIUC to volunteer in a software engineering controlled experiment. We were specifically looking for students that (i) had extensive programming experience, (ii)

Table 5.1: Demographics of the participants.

	Mean	Std.Dev.	Min.	Max
Years Programming	8.35	1.97	5	12
Years Java Programming	4.7	1.72	2.5	7.5
Years Using Eclipse	2.1	1.24	0.5	4

had extensive Java programming experience, and (iii) were familiar with the Eclipse development environment. Table 5.1 shows the distribution of our subject population. Most subjects had some previous industry experience, two of them were active Eclipse committers, another one was a long time industry consultant.

We asked the subjects not to work for more than one hour. For participating in the study, the subjects were rewarded with a free lunch. During the study, the subjects did not know who are other participants and what we are going to do with their code. We told them to implement the task as if this was their regular development job. When we got back their solutions, their implementations were not at all similar.

Tasks Each subject received the initial implementation of the LAN simulation used in our motivating example, along with a passing JUnit test case that demonstrated how to instantiate the system. The system was packaged as an Eclipse project, therefore the subjects had to use the Eclipse development environment. Along with the system, the subjects received a one-page document describing the current state of the system and the new feature they had to implement. We asked them to write another JUnit test case that exercises the feature they just implemented. We also gave them the freedom to change the current design if they did not like it by using the automated refactorings supported in Eclipse. However, only the feature and adding a test case were mandatory, refactoring was optional.

Task ACK required the subject to change the LAN simulation so that when a destination node received a packet, it sends an `Acknowledgement` packet back to the sender. The `Acknowledgement` packet should have its contents set to “ACK”.

Task MAX_HOPS required the subject to fix a problem. In the current implementation the `Packet` may keep on traveling forever if the destination node does not exist. To solve the problem, if a `Packet` has been traveling around long enough without being consumed by any Node, then it gets dumped/eaten up. “Long enough” represents the maximum number of nodes that a `Packet` is allowed to visit and this needs to be specified by the user.

Variables

- **Controlled Variables.** All subjects used Eclipse and Java. Subjects started from the same version and had to implement one of the two tasks. All mergings were done by users expert in CVS and MolhadoRef.
- **Independent Variables.** Merging with MolhadoRef and merging with CVS.
- **Dependent Variables.** Time spent to perform the merging, the number of conflicts that cannot be solved automatically, the number of compile and run-time errors after merging.

5.6.3 Experimental Treatment

Now that we had real data about code developed in parallel, we wanted to merge implementations of ACK and MAX_HOPS into a code base that would contain both features. We use CVS text-based merging as the control group to test whether operation-based merging (with MolhadoRef) is more effective.

After we gathered all the solutions implemented by the subjects, we created pairs as the cross-product among the two groups of tasks (5 solutions for task ACK, 5 solutions for task MAX_HOPS, resulting 25 pairs). Each pair along with the base version of the LAN simulation forms a triplet. For each such triplet, we committed the source code in both CVS (using the Eclipse CVS client) and MolhadoRef. We first committed the base version, then checked out in two different Eclipse projects, replaced the code in the checked out versions with the code for MAX_HOPS and ACK tasks, then we committed the version containing MAX_HOPS task (no merging was needed here), followed by committing the version containing the ACK task (merging was needed here).

By not asking the subjects to do the merging, we prevented them from knowing the goal of our study, so that they would not make subjective changes that could sabotage the outcome of merging. At the same time, we eliminated one of the independent variables that could affect the outcome of merging, namely their experience on merging with CVS or MolhadoRef. Instead, the first and second author (who were both experts with CVS and MolhadoRef) did all the mergings. To eliminate the memory effect, we randomized the order in which pairs were merged.

Table 5.2 shows the results of merging with Eclipse's CVS client versus MolhadoRef.

Table 5.2: Effectiveness of merging with CVS versus MolhadoRef as shown by the controlled experiment.

	Mean	Std.Dev.	Min.	Max
CVS Conflicts	8.04	2	4	11
MolhadoRef Conflicts	2.24	1.23	0	5
CVS Compile Errors	12.08	8.43	0	39
MolhadoRef Compile Errors	1.04	1.09	0	4
CVS Runtime Errors	0.75	1.11	0	5
MolhadoRef Runtime Errors	0.48	0.58	0	2
CVS Time to Merge[mins]	17.5	7.56	8	35
MolhadoRef Time to Merge[mins]	4.96	3.55	1	17

5.6.4 Statistical Results

After applying analysis of variance (ANOVA) using the Paired Student's t-test and Fisher's test, we were able to reject the null hypotheses and accept H1 (MolhadoRef automatically solves more conflicts), H2 (MolhadoRef produces fewer compile-errors), and H4 (it takes less time to merge with MolhadoRef), at a significance level of α = 1%. We were not able to reject the null hypothesis for H3 (MolhadoRef produces fewer runtime errors) at α = 1% level.

5.6.5 Threats to Validity

Construct Validity. One could argue why we chose number of merge errors and time to merge as the indicators for the quality of merging. We believe that a software tool should increase the quality of software and the productivity of programmer. Compile- and run-time errors both measure the quality of the merged code. Number of conflicting blocks measures indirectly how much of the tedious job is taken care by the tool while the time to merge measures directly the productivity of the programmer.

One could also ask why we did all the merging ourselves instead of using the subjects. Partly we wanted to eliminate confounding the effect of tool and the experience of merger, and to control the number of direct variables to a manageable size. We were experts with both CVS and MolhadoRef, where as our subjects would not have any experience with MolhadoRef. In addition, the subjects did not know that their solutions would be merged. This way we simulated an environment where the kinds of changes are not limited by whether or not they can be easily merged, but where programmers have absolute freedom to improve their designs.

Internal Validity. One could ask whether the design of experiment and the results truly represent a cause-and-effect relationship. For instance, since we were the only ones who merged subjects' solutions, the repetition of experiments could have influenced the results. To eliminate the memory effect, we randomized the order in which we merged pairs of solutions. In addition, we split the merging tasks into several clusters, separated by several days. Another question is whether the person who merged with MolhadoRef was better at merging than the person who merged with CVS. Before doing the merging experiment, we tried a few cold-run merging experiments and both persons involved in merging (the first and second author) had the same productivity.

External Validity. One could ask whether our results are applicable and generalizable to a wider range of software projects. We only used one single application and the input code developed in parallel was produced using Eclipse and Java. Maybe by using IDEs which do not feature refactorings, the programmers will make fewer refactorings. Although the presence of refactorings conveniently integrated within an IDE can affect the amount of refactoring, we noticed cases when subjects refactored manually. Also Java is a popular programming language and Eclipse is widely used to develop Java programs.

The subjects of our experiment are all graduate students. On one hand, this is an advantage because the subject demographics are relatively homogeneous. On the other hand, use of students limit our ability to generalize the results to professional developers. However, a careful look at Table 5.1 shows that the subjects had reasonable experience, most of them had worked in industry before coming to graduate school. Notably, two of them had several years of professional consulting and programming experience.

Reliability. The initial base version along with the students' solutions can be found online [Mol], so our results can be replicated.

5.7 Case Study

Most of the development of MolhadoRef was done by two programmers in a pair-programming fashion (two people at the same console). However, during the last three weeks, the two programmers ceased working on the same console. Instead, they worked in parallel; they refactored and edited the source code as before. When merging the changes with CVS, there were many same-line conflicts. It turned out that a large number of them were caused by two refactorings: one renamed a central API class `LightRefactoring` to `Operation`, while the other moved the API class `LightRefactoring` to a package that contained similar

abstractions.

When merging the same changes using MolhadoRef, far fewer conflicts occur. Table 5.3 presents the effectiveness of merging with CVS versus MolhadoRef. Column 'conflicts' shows how many of the changes could not be automatically merged and require human intervention. For CVS these are changes to the same line or block of text. For MolhadoRef these are operations that cannot be automatically incorporated in the merged version because they would have caused compile or run-time errors. Next columns show how many compile-time and run-time errors are introduced by each SCM.

Table 5.3: Effectiveness of merging with CVS versus MolhadoRef as shown by the case study. First rows show how many conflicts could not be solved automatically and required user intervention. Next rows show the number of compile- and runtime-errors after merging with each system. Last rows present the total time (including human and machine time) required to merge and then fix the merge errors.

	MolhadoRef Case Study	LAN Simulation Example
CVS Conflicts	36	3
MolhadoRef Conflicts	1	1
CVS Compile Errors	41	1
MolhadoRef Compile Errors	0	0
CVS Runtime Errors	7	1
MolhadoRef Runtime Errors	0	0
CVS Time to Merge [mins]	105	5
MolhadoRef Time to Merge [mins]	2	1

Table 5.3 shows that MolhadoRef was able to automatically merge all 36 same-line conflicts reported by CVS. MolhadoRef asked for user assistance only once, namely when both developers introduced method `getID()` in the same class. MolhadoRef did not introduce any compile-time or run-time errors while CVS had 48 such errors after "successful" merge. In addition, it took 105 minutes for the two developers to produce the final, correct version using CVS, while it takes less than two minutes to merge with MolhadoRef.

5.8 Summary

Upgrading component-based applications to use the latest version of component can be seen as a special case of software merging. The upgrading tool needs to merge refactorings and edits on the component side, with those on the application side. In addition, it needs to identify the merge conflicts that require user

intervention.

Because refactorings can potentially change many parts of a program, they create problems for current text-based SCM tools. These SCM systems would signal merge conflicts that can be automatically resolved by a refactoring-aware merging algorithm. In addition, they do not signal semantic conflicts, but only syntactic conflicts in cases when the changes are on the same lines of code.

We present a novel SCM system, MolhadoRef, that is aware of program entities and the refactoring operations that change them. MolhadoRef uses the operation-based approach to record (or detect) and replay changes. By intelligently treating the dependences between different operations, it merges edit and refactoring operations effectively. In addition, because MolhadoRef is aware of the semantics of change operations, a successful merge does not produce compile or runtime errors.

The reader can find screenshots and download MolhadoRef at [Mol].

Chapter 6

Refactoring-aware Binary Adaptation of Evolving Libraries

6.1 Introduction

Our previous chapters, as well as most previous solutions to automate upgrading of clients advocate that (a) source code of clients [CN96, RH02, HD05, BTF05] or (b) bytecode [KH98, WSM06] of clients be changed in response to library API changes. However, this is not always possible. For example, Eclipse IDE [Ecla] ships with many 3rd party clients that extend the functionality of the IDE. These clients are distributed in bytecode format only, without source code, and, for legal reasons, have software licenses that prohibit changing the bytecodes of clients.

Rather than waiting for 3rd party client developers to update their clients whenever the library APIs change, we propose that clients are automatically *shielded* from the API changes in the library. Our solution restores the *binary compatibility* of the library: older binary clients continue to link and run against the new libraries without changing or recompiling the clients. It automatically generates a *refactoring-aware compatibility layer* using the refactoring trace recorded by the refactoring engine used by the library developers; alternatively, the trace can be inferred using our previous tool, RefactoringCrawler [DCMJ06], thus no need to manually annotate libraries. This compatibility layer automatically restores library's API compatibility while giving library developers the freedom to improve the library APIs.

This chapter presents *ReBA*, our refactoring-aware binary adaptation tool. ReBA has several distinguishing features that make it practical for modern large software systems comprised of core libraries and many 3rd party clients. We designed ReBA to meet the criteria that we believe are required for modern systems:

- **Binary clients:** the solution should work with the binary version of the clients. This is desirable because large systems (e.g., IDEs) often do not own the source code of 3rd party clients that extend the functionality of the system.

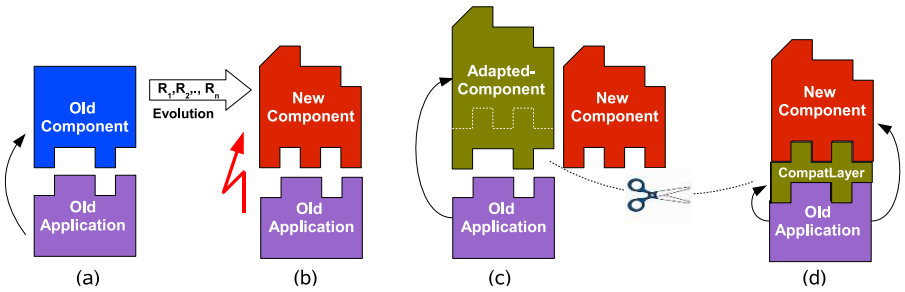

Figure 6.1: Overview of our approach (arrows show dependences between modules). (a) Library and client are compatible. (b) API changes in library break compatibility. (c) ReBA generates an adapted-library that supports both the old and the new APIs. (d) From the adapted-library, ReBA carves out a client-specific compatibility layer. The old client loads the restored APIs from the compatibility layer and the APIs not changed from the new library.

- **Dual Support:** Consider a large system like the Eclipse IDE that ships with core libraries and 3rd party clients. Some clients are updated regularly to use the new versions of libraries, but others are not. A solution must allow both old and updated clients to simultaneously use the new version of the libraries and be included in the shipped product.

- **Preserve Edits**: the proposed solution should preserve the edits (e.g., performance improvements or bug fixes) of the library. This is important because old clients can benefit from the enhancements in the library.

- **Preserve Type Information**: A known problem with compatibility layers using the Adapter pattern [GHJV95](also known as Wrapper) is that object identity breaks, since adapter and the adaptee are two different classes. We want to ensure that old clients that use type, identity, or equality comparisons (e.g., `instanceof`, `==`, or `equals`) would still work.

Figure 6.1 shows an overview of our solution. ReBA works in two main stages. First, it creates a version of the library, called *adapted-library* (details in Section 6.4). The adapted-library contains the same code as the new version of the library, but it supports both the old and the new APIs. Second, from this adapted-library ReBA carves a custom-made, client-specific *compatibility layer* (details in Section 6.5) using a static analysis technique based on *points-to* analysis [Wei80].

This technique greatly reduces the size of the compatibility layer, while our conservative static analysis ensures that an old client loads at runtime only classes that are backwards compatible. The compatibility

layer allows both old and new clients to run simultaneously with the refactored library. In addition, since the compatibility layer uses the same class names that the old client expects, the layer preserves the type information and identity of the adapted classes.

This chapter makes three major contributions:

1. An innovative solution. Our solution automatically generates a compatibility layer for a library that allows both the old and new interface to be used in the same system. It does not require programmers to annotate their library, but builds the compatibility layer using information recorded by a refactoring tool. It uses points-to analysis in a new way to minimize the size of the layer.

2. Implementation. ReBA, our concrete implementation works for Java programs. ReBA is implemented as an Eclipse plugin, therefore it is conveniently integrated into the widely used Eclipse IDE. To minimize the size of the compatibility layer, we implemented the points-to analysis using WALA [WAL], an analysis library from IBM Research.

3. Evaluation. We have used ReBA to generate compatibility layers in different scenarios (see Section 6.7). First, we evaluated ReBA on the data obtained from a controlled experiment with 10 different developers. Second, we evaluated ReBA on three Eclipse libraries which were refactored by Eclipse developers. For both scenarios we used ReBA to generate and apply compatibility layers to a comprehensive test suite created for a pre-refactoring version of the library. After applying our compatibility layers, the tests ran successfully. Profiling shows that the memory and running overhead imposed by our solution is small.

ReBA and all our experimental results are publicly available online from the ReBA web page:
`http://netfiles.uiuc.edu/dig/ReBA`

6.2 Motivating Examples

We show three examples of API changes that cause problems for clients. These examples are taken from real world case-studies, namely libraries that make up the official distribution of Eclipse IDE [Ecla].

With respect to how ReBA generates the compatibility layers, API changes can be separated into three categories which are mutually exclusive and cover all cases: deletions of APIs, method-level API refactorings, and refactorings that change the names of classes (details in Section6.4). Each of the examples below belongs to one of the three categories. The first three examples illustrate the need for the adapted-library, the fourth example illustrates the need to reduce the size of the adapted-library.

```
// version 20070329 before refactoring
class RefactoringExecutionStarter {
  ...
  void startCleanupRefactoring(
                ICompilationUnit[] cus,
                boolean showWizard, Shell shell) {

    ICleanUp[] cleanUps= CleanUpRefactoring.
        createCleanUps();
    for (int i= 0; i < cleanUps.length; i++)
       refactoring.addCleanUp(cleanUps[i]);
    ...
  }
}
```

```
// version 20070405 after refactoring
class RefactoringExecutionStarter {
  ...
  void startCleanupRefactoring(
            ICompilationUnit[] cus,
            boolean showWizard, Shell shell,
            ICleanUp[] cleanUps,
            String actionName) {

    for (int i= 0; i < cleanUps.length; i++)
       refactoring.addCleanUp(cleanUps[i]);
    ...
  }
}
```

Figure 6.2: Example of ChangeMethodSignature refactoring in plugin `jdt.ui`. Two new arguments were added: `cleanUps` was previously a local variable and is now extracted as an argument, `actionName` is a brand new argument.

Deletion of APIs. In library `jface.text` version 20060926 (format yearmonthday), class `DiffApplier` is deleted. There is at least one client, `workbench.texteditor.tests` that uses this class. As a result of the deletion of `DiffApplier`, any version of this client prior to 20060926 no longer works with the newer version of the library. To make this change backwards compatible, ReBA creates a version of the library, which we call adapted-library, where `DiffApplier` is added back. Other than adding a dependency between the test client and the adapted-library, ReBA requires no changes to the client.

Method-level refactorings. In library `jdt.ui` version 20070405, the signature of method `RefactoringExecutionStarter.startCleanupRefactoring` changed from three arguments to five arguments (see Fig.6.2). The library developers provided default values for the two new arguments such that calling the 5-argument method with the default values would behave like the previous version with 3 arguments.

There is at least one client broken because of this API change, the testing client `jdt.ui.tests`. To shield this client from the change, ReBA generates an adapted-library containing a version of `jdt.ui` where the original method with 3 arguments is restored. The method merely delegates to the new version with 5 arguments, and passes the default values for the extra arguments (see Fig.6.3). Because the adapted-library delegates to the latest implementation of the method, old clients can benefit from the improvements in the library. When supplying the compatibility layer to the old client client, all tests run successfully.

Refactorings that change Class names. In library `workbench.texteditor` version 20060905, class `Levenstein` was renamed to `Levenshtein` (notice an extra "h" character). There is at least one client, `workbench.texteditor.tests` that is broken because of this change. ReBA generates an adapted-library

```
1
2  //Adapted Library Class
3  class RefactoringExecutionStarter {
4    ...
5    // ReBA adds this method with the old signature
6    void startCleanupRefactoring(ICompilationUnit[] cus,
7                  boolean showWizard, Shell shell) {
8      return startCleanupRefactoring(cus, showWizard, shell,
9             CleanUpRefactoring.createCleanUps(),"Clean Up");
10   }
11
12   //new method with 5 arguments
13   void startCleanupRefactoring(ICompilationUnit[] cus,
14            boolean showWizard, Shell shell,
15            ICleanUp[] cleanUps, String actionName) {
16     //same implementation as in the
17     //new version of the library
18     ...
19   }
20 }
```

Figure 6.3: Compatibility layer generated for ChangeMethodSignature refactoring in library `jdt.ui`.

where the class name `Levenshtein` is reverted back to `Levenstein` (all other code edits in `Levenshtein` are preserved). Because the adapted-library consistently uses the same class names that the old client expects, client code that uses type checking (e.g., `instanceof`) or makes use of object equality (`equals`) or object identity (==), still runs as before, without requiring any changes to the client.

The need for carving: After creating the adapted-library, in order to enable both upgraded and old clients to run simultaneously, one needs to keep both the adapted-library and the new version of the library. This can result in doubling the memory consumption.

To alleviate this problem, once it generated the adapted-library, ReBA carves out a client-specific compatibility layer which is much smaller than the adapted-library. Recall our previous example where ReBA generated the adapted-library by reverting the renamed class `Levenshtein` back to `Levenstein`. Figure 6.4 shows some classes in the adapted-library that use `Levenstein`. A client can use `Factory` to create instances of `Levenstein` class. `Factory` has a direct source reference to `Levenstein`, therefore `Factory` needs to be in the compatibility layer. If it was not in the compatibility layer, the client would load the version of `Factory` from the new library where `Factory` creates instances of `Levenshtein`, thus breaking the client.

The other library class, `Indirect`, does not appear to have a reference to `Levenstein`. Does

```
1  class Factory {
2    Levenstein create(){
3      return new Levenstein();
4    }
5   }
6
7  class Indirect {
8    Object m() {
9      Factory factory = new Factory();
10     return factory.create();
11   }
12  }
```

Figure 6.4: Library classes that use `Levenstein`.

`Indirect` needs to be put in the compatibility layer? Yes. Although the source code of `Indirect` does not have a reference to `Levenstein`, its bytecode does. This is because the method call to `create` in the bytecode of `m` is replaced by the method descriptor for `create`. Bytecode method descriptors contain not just the method name and arguments, but also the return type [LY01]. Therefore, in order to restore the backward compatibility, the version of `Indirect` that is compiled with `Levenstein` needs to be put in the compatibility layer, otherwise a runtime error "method not found" is thrown. ReBA uses a points-to analysis (details in Section6.5) to find out all classes in the adapted-library that can refer to `Levenstein`, thus it finds both `Factory` and `Indirect` and adds them to the compatibility layer. The carved compatibility layer contains only `Levenstein`, `Factory`, and `Indirect`.

6.3 Overview

6.3.1 Background Information

During library evolution, most changes add new API methods and classes that do not affect old clients. Therefore, we distinguish between API changes that are backwards compatible (e.g., additions of new APIs), and those that are not backwards-compatible (e.g., deletion of API methods, renaming API classes). Since only the latter category affects old clients, from now on, by API changes we refer to those changes that are not backwards compatible.

Besides refactorings, libraries evolve through edits. We distinguish between *API edits* (e.g., edits that change the APIs) and *code edits* including all remaining edits (e.g., performance improvements, bug fixes).

Code edits in general have less defined semantics, making it harder for tools to automatically reason about them. ReBA preserves in the adapted-library all code edits from the latest version of the library, and adapts the API deletions and API refactorings. Refactorings preserve the semantics, but edits do not. Although ReBA intends to ensure that old binary clients run "correctly" with the new versions of the library, because of the edits, we cannot give such strong semantic guarantees.

In the current implementation, ReBA supports these refactorings: rename package, rename class, rename method, rename field, move method to different class, change method signature, encapsulate field with accessor methods. These refactorings are some of the most frequently occurring refactorings in practice [DJ06]. The API edits that ReBA currently supports are: delete package, delete class, and delete method.

6.3.2 High-level Overview

This section gives an overview of each of the two stages of our solution.

Creating the Adapted Library One approach to creating a backwards-compatible, adapted-library, is to start from the new version of the library and undo all API changes, in the reverse order in which they happened. Although simple, this approach has two limitations. First, one needs to selectively undo refactorings while ignoring API additions. There can be dependences between refactorings and API additions, so undoing one without the other is hard. Second, from a practical point of view, tools that record API changes might not store all the information required to undo an API change. For example, Eclipse refactoring logs do not store enough information to undo deletions.

Therefore, ReBA does not undo API changes on the newer version of the library, but it replays the (modified) API changes on the old version of the library. When replaying the API changes, ReBA alters them such that they preserve the backwards compatibility (e.g., a delete operation is not replayed, and a rename method leaves a delegate).

Figure 6.5 shows an overview of creating the adapted-library. The algorithm takes as input the source code of the new and old versions of the library, and the trace of API changes that lead to the new version. These API changes can be retrieved from an IDE like Eclipse (version 3.3) that automatically logs all refactorings and deletions. Alternatively, API changes can be inferred using our previous tool Refactor-

INPUT: $Library_{New}$, $Library_{Old}$, $\Delta_{Library} = (R_1, R_2, ..., R_n)$
OUTPUT: $AdaptedLibrary$
Creating AdaptedLibrary **begin**
1 $AdaptedLibrary$ = $Library_{New}$
2 **forEach**(Operation op: $\Delta_{Library}$)
3 Operation $\widetilde{op}$ = `preserveOldAPI(op)`
4 $Library_{Old}$ = `replayOperation`($\widetilde{op}$, $Library_{Old}$)
5 **endForEach**
6 $AdaptedLibrary$ += `copyRestoredElements`($Library_{Old}$)
7 $AdaptedLibrary$ = `reverseClassNames`($AdaptedLibrary$)
end

Figure 6.5: Overview of creating the Adapted-Library.

ingCrawler [DCMJ06].

ReBA starts from the old library version and processes each library API change in the order in which they happened. For each API change op, ReBA creates a source transformation, $\widetilde{op}$, that creates the refactored program element and also preserves the old element. Recalling our example in Fig.6.2, ReBA creates another change signature refactoring which keeps both the old and the new method (the old method merely delegates to the new method). Then ReBA replays the $\widetilde{op}$ transformation on the old version of the library.

Once it processes all API changes, ReBA copies the changed program elements from $Library_{Old}$ to $AdaptedLibrary$. Although the $AdaptedLibrary$ starts as a replica of the new library version, with each copying of program elements from the modified $Library_{Old}$, it supports more of the old APIs.

The compatibility of classes with changed names cannot be restored by copying, but only by restoring their names. This is done by reversing refactorings that changed class names. For example, ReBA restores the API compatibility of `Levenshtein` in our motivating example by renaming it back to `Levenstein`.

At the end, the adapted-library contains all the APIs that the old client requires. The algorithm processes refactorings from different categories differently. Section 6.4 presents one example from each category.

Carving the Compatibility layer To optimize for space consumption, ReBA constructs a client-specific compatibility layer. The customized compatibility layer consists of classes whose API compatibility was restored in the adapted-library, as well as those classes that are affected by this change. Some classes are *directly affected*, for example, a class that has a reference to a renamed class. Other classes are *indirectly affected*, for example classes that do not have a source reference, but only a bytecode reference to the

INPUT: $AdaptedLibrary$, $client$, $\Delta_{Library} = (R_1, R_2, ..., R_n)$
OUTPUT: $CompatLayer$
Carving the Compatibility Layer **begin**
1 $CompatLayer$ = Φ
2 Graph pointsToGraph =
3 `buildPointsToGraph`($AdaptedLibrary, client$)
4 **forEach**(Operation op: $\Delta_{Library}$)
5 Class[] classes= `getChangedClassesFrom`(op)
6 Set reachingClasses =
`getReachingNodes`(pointsToGraph, classes)
7 $CompatLayer$ = `append`($CompatLayer$, {classes, reachingClasses})
8 **endForEach**
end

Figure 6.6: Overview of carving the compatibility layer.

restored class. Recalling our example from Fig. 6.4, ReBA needs to put in the compatibility layer both `Factory` (directly affected) and `Indirect` (indirectly affected), in order to restore the binary compatibility.

To find both directly and indirectly affected classes, and to keep only those classes that a specific client can reach to, ReBA uses a *points-to* analysis. Points-to analysis establishes which pointers, or heap references, can point to which variables or storage locations. Figure 6.6 gives an overview of how we use points-to analysis to determine the direct and indirect affected classes, the details of the analysis are found in Section 6.5.

Starting from the client, ReBA first creates a directed graph (see Fig.6.6) whose nodes are library classes reached from the client and whose edges are points-to relations. Then ReBA iteratively processes each API change and determines the classes that are changed due to restoring their backwards compatibility. ReBA traverses the $pointsToGraph$ and gets all other library classes that can reach to the restored classes. For each refactoring, the compatibility layer grows by adding the classes that are changed as well as their reaching classes.

Putting it all together Next we show how an old client is using the compatibility layer along with the new version of the library. First we give a gentle introduction to how classes are loaded in Java. ReBA does not require any special class loading techniques, thus it enables anybody who uses the standard class loading to benefit from our solution.

In Java, classes are loaded lazily, only when they are needed. Classes are loaded by a *ClassLoader*.

When running a client, one has to specify the *ClassPath*, that is the places where the bytecodes of all classes are located. The ClassPath is a sequence of classpath entries (e.g., Jar files, folders with bytecode files), each entry specifying a physical location. When loading a class, the ClassLoader searches in each entry of the classpath until it can locate a class having the same fully qualified name as the searched class. In case there are more than one class that match the fully qualified name, the ClassLoader loads the first class that it finds using the order specified in the class path entries.

After creating the compatibility layer, ReBA places it as the first entry in the ClassPath of the client. Thus, at runtime, when loading a class, the ClassLoader searches first among the classes located in the compatibility layer.

Suppose that the old client asks for a class whose API compatibility was restored by ReBA. The ClassLoader finds this class in the compatibility layer and it loads it from there. Even though a class with the same name (but which is not backwards compatible) might exist in the new library, because the ClassPath entry for the compatibility layer comes before the entry for the new library, the ClassLoader picks the class from the compatibility layer.

Now suppose that the client searches for a class that is backwards compatible even in the new version of the library and is not present in the compatibility layer. The ClassLoader searches for this class in the compatibility layer; it is not found here, and the ClassLoader continues searching in the new library where it finds the required class.

Now suppose that we have both an old client and a new client running altogether. The compatibility layer allows the old client to load classes that are backwards compatible as described above. As for the new client, since it does not have the compatibility layer among its ClassPath entries, it can only load classes from the new library.

6.4 Creating the Adapted Library

This section describes how the adapted-libraries are generated by presenting one example from each of the three categories of API changes. We use the same examples as in our motivating section (Sec. 6.2). This section presents one example from each category of API changes.

Delete API class, delete method, and delete package form the Deletion of API category (Subsection 6.4.1). Rename method, change method signature, move method, replace field references with access

method belong to Method-level category (Subsection 6.4.2). Rename class, move class, rename package, and move package belong to the category that changes the fully qualified names of classes (Subsection 6.4.3).

6.4.1 Deletion of APIs

For operations belonging to this category, function `preserveOldAPI` (line 3 in Fig.6.5) returns a `NOP` operation. For example, when processing the operation that deletes class `DiffApplier`, ReBA does not replay the deletion, thus it keeps `DiffApplier` in the old version of the library. Later, function `copyRestoredElements` copies `DiffApplier` from the old version of the library to the adapted-library.

Function `reverseClassNames` does not process operations belonging to this category, since their API compatibility was restored previously by function `copyRestoredElements`.

6.4.2 Method-level Refactorings

For operations belonging to this category, function `preserveOldAPI` returns a new operation. The new operation is the same kind of refactoring as the original refactoring. However, in addition to replacing the old method with the refactored method, it creates another method with the same signature as the old method, but with an implementation that delegates to the refactored method.

For our motivating example of ChangeMethodSignature (Fig. 6.2) that adds two arguments to `startCleanupRefactoring`, the new operation produced by `preserveOldAPI` is another ChangeMethodSignature refactoring. Because ReBA is implemented on top of Eclipse, it uses the Eclipse representation to create new refactorings. In Eclipse (version $\geq$ 3.2), when playing a refactoring, the refactoring engine logs each refactoring and it produces a refactoring descriptor. This refactoring descriptor is a textual representation of the refactoring and contains information such as: the type of refactoring, the program element on which the refactoring operates (identified via a fully qualified name), and arguments supplied by the user through the UI (e.g., the new name in case of a rename refactoring).

Function `preserveOldAPI` takes a refactoring descriptor and it creates another refactoring descriptor. In our example, the new refactoring descriptor specifies the same properties as the old descriptor: the type of refactoring, the input element `startCleanupRefactoring`, the default values for the two extra arguments (the String literal “Clean Up” and “CleanUpRefactoring.createCleanUps()”), etc. In addition, ReBA adds

another tag specifying that the original method should be kept and delegate to the refactored method. Out of the new descriptor, ReBA creates a refactoring object and replays this refactoring on the old version of the library.

Figure 6.7 shows that for the motivating example (Fig. 6.2), function `copyRestoredElements` copies only the restored method with the old signature from the refactored version of the old library to the adapted-library. ReBA does not copy the method with the new signature from the old library. Recall that by construction, the adapted-library contains all code edits from the new version of the library. Thus, the adapted-library uses the same implementation of this method as the new version of the library. After copying the old method in the adapted-library, because the copied method delegates to the new implementation of the method (located in the same class), ReBA allows clients to benefit from the improvements (e.g., bug fixes, performance improvements) in the new version of the library.

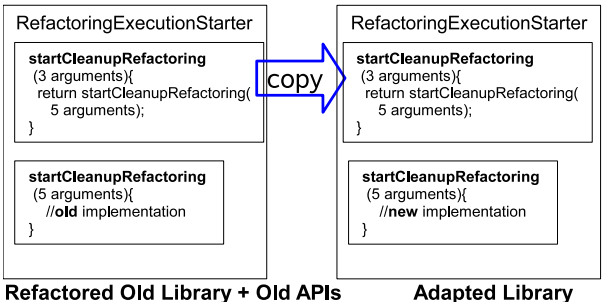

Figure 6.7: Using the ChangeMethodSignature motivating example (Fig. 6.2), function `copyRestoredElements` copies only the restored API elements.

Function `reverseClassNames` does not process operations from this category since their API compatibility was restored previously.

6.4.3 Refactorings that Change Class Names

Operations in this category include all those that change the fully qualified (e.g., package.class) names of classes. Fully qualified names play a central role in how classes are loaded at runtime; any changes to these names would cause older clients to fail to load the class. Therefore, restoring the compatibility of a class whose name has changed, means restoring the name of the class. This restoration happens in two steps.

For refactorings belonging to this category, function `preserveOldAPI` is the identity function, e.g.,

returns back the original refactoring. Although this operation does not restore the backwards compatibility, ReBA still replays it on the old library. This is needed because later refactorings could depend on the names changed by refactorings in this category. More specifically, later refactoring cannot be executed unless the previous refactorings were executed. The reason for the dependences between refactorings is the fact that refactoring engines identify program elements by their fully qualified names (e.g., package.class.methodName for a method). For example, given the sequence:

$\Delta_{Library}$ = {op_1= RenameClass($A \rightarrow B$);
op_2 = MoveMethod($B.m \rightarrow C$)}

Refactoring op_2 could not discover method `B.m` unless it looks into class `B`, thus the renaming must take place before op_2 could proceed.

Function `copyRestoredElements` does not copy elements affected by this kind of refactorings, because these elements are not yet backwards compatible. Function `reverseClassNames` is the one that restores the compatibility of these program elements by creating a *reverse* refactoring and applying it on the adapted-library. In our motivating example, function `reverseClassNames` reverts `Levenshtein` back to `Levenstein` by applying a reverse rename class refactoring.

To restore all class names, ReBA creates reverse refactorings for all refactorings that change class names. Then ReBA applies the reverse refactorings backwards, from the last operation in $\Delta_{Library}$ toward the first operation.

ReBA creates the reverse refactoring by constructing a new refactoring descriptor where it swaps the old and new names of the class. From this refactoring descriptor, ReBA creates a refactoring object and then uses the Eclipse refactoring engine to apply the refactoring.

6.5 Carving the Compatibility Layer

From the adapted-library, ReBA carves a smaller, client-specific compatibility layer. This layer contains only the classes whose API compatibility was previously restored and those classes from which a client could reach/load the restored classes. All the remaining classes need not be present in the compatibility layer, and can be loaded from the new version of the library.

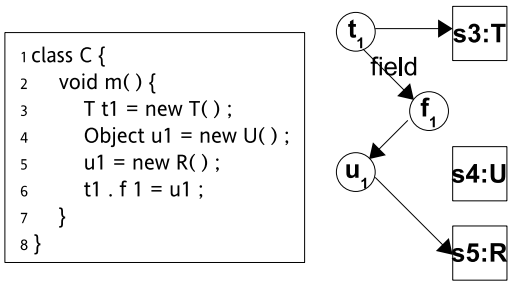

Figure 6.8: An example points-to graph.

6.5.1 Points-to Analysis

The heart of this stage is the use of *points-to* analysis to find the set of classes, *Reaching*, that can "reach" to classes whose API compatibility is restored. Points-to analysis establishes which pointers, or heap references, can point to which variables or storage locations. Figure 6.8 illustrates the creation of the points-to graph for a simple program.

In Fig. 6.8 allocated objects are marked with squares ($< s_i : T >$) where i shows the line number where the object is created, T represents the type). Pointers to objects (e.g., references in the Java terminology) are denoted by named circles, having the same name as the pointer. Directed edges connect pointers to the objects they point to, or to other pointers (e.g., field f1 points to pointer u1). Fields are also connected to their objects by directed edges.

The points-to analysis that we use is flow-sensitive: it takes into account the order of statements (e.g., line 5 "kills" the previous points-to relation $u1 \rightarrow s_4 : U$). The analysis is also field-sensitive: each instance of a field is modeled with a separate variable. The points-to analysis is conservative, meaning that at runtime some pointers "might" point to the computed allocated objects. Having false positives means that our analysis can add unnecessary classes to the compatibility layer. However, it is much more important that the analysis does not miss cases where a class should be added to the compatibility layer.

To build the points-to graph, we use WALA [WAL], a static analysis library from IBM Research. The example we illustrated shows the points-to graph for an *intraprocedural* analysis. When the function has calls to other functions, WALA does an *interprocedural*, context-sensitive analysis (e.g., it takes the calling context into account when analyzing the target of a function call.) To compute an interprocedural analysis, WALA first creates the call graph using the *entry-points* (e.g., main methods) from the client. WALA does

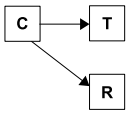

Figure 6.9: Example of points-to relations between classes used in Fig.6.8.

not need the source-code of the client or library, it only needs the bytecode of client and library. Using the information from the call graph, the interprocedural analysis expands a function calls. In other words, it builds the points-to graph by traversing "through" function calls.

From the points-to graph created by WALA, ReBA creates a customized points-to graph where it represents points-to relations at a higher granularity. Specifically, the custom points-to graph represents points-to relations only among classes. First, it short-circuits the original points-to graph such that each pointer points to a leaf object (e.g., it changes $t_1 \rightarrow f_1 \rightarrow u_1 \rightarrow s_5 : R$ to $t_1 \rightarrow s_5 : R$). Next, for each points-to relation in the short-circuited graph it creates a points-to relation between the class where the pointer is defined and the class of the pointed object. For the graph in Fig. 6.8, ReBA creates the points-to graph in Fig. 6.9 whose nodes are classes and the edges are points-to relations between classes.

Next, for each API change, ReBA determines the set of classes changed in order to restore the API compatibility. Then ReBA searches in its points-to graph for all classes that can "reach" to these changed classes through the points-to relations. Finally, ReBA puts the *Reaching* classes and the changed classes in the custom-made compatibility layer.

6.6 Discussion

During the first stage when ReBA generates the adapted-library by replaying the modified API changes, ReBA only needs access to the source code of the library, not the client. Since the library developers use ReBA to generate the adapted-library, ReBA has no problem accessing the library source code. With each new library version, library developers can ship a signed, trusted adapted-library that is backwards compatible. Client developers can use ReBA to carve out a custom compatibility layer. Additionally, library developers could also use ReBA to generate custom compatibility layers for clients for which they do not have access to the source code. For example, large frameworks like Eclipse ship with 3rd party clients for which the framework providers do not have access to source code. Eclipse developers could use ReBA to

accommodate 3rd party clients that make up the official distribution of Eclipse.

In Java, the bytecode of the client contains as much information as the source code, essentially it is just another representation. In cases when the source code of clients is missing, one could ask why not change the bytecodes of clients directly? There is no technical reason why one could not refactor the bytecodes of the client in response to library refactorings. However, there might be several other reasons why changing the bytecodes is not an option. First, for legal reasons, several proprietary companies and software licenses forbid any changes in the bytecodes of shipped clients. Even some open-source licenses only allow changes in the source code. Second, even the smallest change in the source code of clients might require recompilation, thus rendering the previous bytecode patches invalid. Having a bytecode patch that needs to be recreated and reapplied at each recompilation of clients can easily lead to expensive maintenance. Third, the bytecodes could be intentionally obfuscated for privacy reasons.

Our approach to adapt libraries is not limited to two consecutive versions of the library, but works for any two versions of a library. Given three different library versions, V_1, V_2, and V_3, the adapted-libraries/compatibility layers generated are not stacked on top of each other, but each adapted-library/compatibility layer works directly with the latest version of the library, thus reducing compositional performance overhead.

To generate a custom-compatibility layer, rather than using our points-to analysis, the naive approach is to search for all classes that have a source code reference to the adapted classes and to put these referencing classes in the compatibility layer. However, this solution has several disadvantages: (i) it requires availability of source code, (ii) it fails to identify classes that are referenced indirectly (see example `Indirect` in Fig. 6.4), (iii) it adds unnecessary classes. In the example illustrated in Fig. 6.8- 6.9, if class `U` is refactored, the naive solution would add class `C` to the compatibility layer, because it has a direct reference to the refactored class. This is unnecessary, because class `C` cannot "escape" `U` outside of its `m` method, thus a client can never get hold of class `U` from class `C`.

A different alternative to points-to-analysis, is to scan the bytecodes of the adapted-component and only put in the compatibility-layer those classes whose bytecodes have changed. This approach would work in a traditional Java application which uses one single class loader. However, it would not work when an application uses multiple modules, each of them having its own classloader and classpath, like in the Eclipse IDE plugin architecture. ReBA needs to put in the compatibility layer even some classes that have

not changed. The use of points-to analysis is instrumental in finding those classes that do not change at all, still they transitively refer to the restored classes. Our solution works for both the traditional scheme with one class loader per application, and the more advanced schemes with several class loaders per application.

Although ReBA does not currently handle all refactorings, we examined all refactorings supported by the Eclipse IDE and they all could be adapted in a similar way with the one described in this chapter. Some Eclipse refactorings are composed of smaller refactorings. For example, ExtractClass refactoring which introduces another class in the type hierarchy and moves methods in the introduced class can be implemented by combining refactorings from our second category (method-level) and refactorings from the third category (class names). Changes to the type hierarchy need to be reverted in a similar way how we revert class names.

Although our solution is not language-specific, its implementation is. In order to adapt our solution to other programming languages, these are the areas which will need to be changed. First, class lookup mechanism is language-specific. In Java, we place the compatibility layer as the first entry in the Classpath. For example, Smalltalk uses a dictionary to lookup classes, thus the adapted classes need to be placed in the class dictionary. Second, the points-to analysis uses the WALA library for Java, and an equivalent library will be needed for the new programming language. A language-specific points-to analysis can be used to ensure that all class libraries that can "escape" the adapted-classes are placed in the compatibility layer in order to preserve the type and information identity of objects. Third, refactoring engines are language-specific, so Eclipse's refactoring engine needs to be replaced by another engine.

Strengths: Our solution meets all four practical criteria outlined in Section 6.1: it does not need access to the source code of clients, nor does it require any changes to the clients. Second, it supports both old and new clients to run against a new version of the library. Third, it preserves the edits in the library. Fourth, it preserves the type and identity of the adapted classes.

The current state-of-the-practice in library design favors deprecating the old APIs, and removing them after several iterations. However, deprecated APIs are rarely removed. For example, Java 1.4.2 runtime library has 365 deprecated methods. Our solution enables libraries to evolve without resulting in the proliferation of APIs. ReBA enables library developers to maintain crisp APIs at the source code level, while at the binary level, the old APIs are added back automatically for compatibility reasons.

Limitations: The key aspect of our solution is that instead of putting a wrapper around the library, we give

it two interfaces, an old API and a new API. This only works when the two interfaces are compatible. It is fine for the interfaces to intersect, but they can not contradict each other. For example, if library developers delete method `A.m1`, and then rename `A.m2` → `A.m1`, it is not possible to support both the old and the new interface of class `A`. Although this might seem like a severe limitation of our approach, in practice we never met such conflicting changes.

Our adaptation approach adds some memory and CPU overhead. However, the evaluations show that this overhead is small enough to be considered acceptable.

6.7 Evaluation

The goal of the evaluation is two-fold: (i) we want to assess the effectiveness of our solution, and (ii) we want to assess the efficiency of our solution. More specifically, we want to answer the questions:

- **Q**1(effectiveness): Does our generated compatibility layer allow older clients to run with newer versions of libraries?
- **Q**2(efficiency): Is the generated compatibility layer only as large as a concrete client would need? What is the performance overhead imposed by our solution?

To answer these questions we conducted different studies. The first study is a controlled experiment of 10 developers evolving a LAN library [DRG+05] independently. The second study is a suite of case-studies using ReBA to restore the API compatibility of Eclipse core libraries. We used comprehensive JUnit test suites developed by the library provider. If the old tests succeed with the new version of the library, it implies a strong likelihood that a regular client application would succeed too.

6.7.1 Controlled Experiment

We asked 10 developers to independently evolve a version of a LAN simulation library [DRG+05]. The library code was accompanied by an automated JUnit test suite. Each developer had to implement one of two features. How they implemented the feature was their choice. In addition, developers had the freedom to refactor any APIs in the library, though we did not influence their choices. We requested that the developers not work for more than one hour.

Table 6.1: Demographics of the participants.

	Mean	Std.Dev.	Min.	Max
Years Programming	8.35	1.97	5	12
Years Java Programming	4.7	1.72	2.5	7.5
Years Using Eclipse	2.1	1.24	0.5	4

Table 6.2: The effectiveness and efficiency of ReBA compatibility layers for the controlled experiment before/after applying the compatibility layer.

Developer	API-Breaking	CompileErrors		TestFailures		MemoryUsed[B]		Memory	RunningTime[ms]		Running
Solution	Changes	Before	After	Before	After	Before	After	Overhead	Before	After	Overhead
#1	1 ChangeMethodSignature, 6 EncapsulateField	34	0	3	0	1418312	1532640	8.06%	14.37	14.02	-2.39%
#2	6 EncapsulateField	29	0	3	0	1417896	1532216	8.06%	14.98	14.35	-4.21%
#3	2 ChangeMethodSignature	4	0	2	0	1414864	1533584	8.39%	9.65	10.03	3.93%
#4	1 RenameMethod, 2 ChangeMethodSignature	2	0	1	0	1418648	1535296	8.22%	8.7	8.9	2.65%
#5	2 RenameType, 1 RenameField	19	0	3	0	1420400	1526888	7.49%	7.86	7.99	1.56%
#6	3 RenameMethod	5	0	2	0	1422376	1538616	8.14%	10.0	10.1	1.29%
#7	6 EncapsulateField	29	0	3	0	1419296	1532056	7.94%	14.1	13.7	-2.33%
#8	1 DeleteMethod, 1 EncapsulateField	6	0	3	0	1421480	1532968	7.84%	11.4	11.0	-2.98%
#9	3 ChangeMethodSignature	2	0	2	0	1426664	1534736	7.57%	7.94	8.26	4.01%
#10	0	0	0	0	0	1358884	1358884	0	7.62	7.62	0

Demographics: Although the developers were graduate students, they were mature programmers with several years of programming experience. Most of them had worked in industry previously (2 of them had extensive consulting experience, two others were active Eclipse committers). Each developer implemented successfully the required feature. Table 6.1 shows the demographics of our population.

Experimental treatment: We took their solutions, and used ReBA to generate a compatibility layer for each of their solution so that we could run the original JUnit tests.

Results: Table 6.2 shows the number of compile errors and test failures when putting together the old JUnit test suite with the new library. Each row displays the data for a particular developer solution: the number of errors before and after placing the ReBA-generated compatibility layer. After using the ReBA-generated compatibility layers, there were no problems for any the 10 solutions. The last row shows a solution that did not contain any refactoring nor other API-breaking changes - as a consequence, ReBA generated an empty layer and the original test ran successfully.

To answer the efficiency question, with regards to the minimality of our compatibility layers, we manually examined the generated compatibility layers and they contain exactly the classes that the old tests need. Removing any class from the compatibility layer would break the API compatibility.

We used the TPTP [eclipse.org/tptp] profiler to measure the memory and the runtime overhead of using ReBA compatibility layers. To measure the overhead, for each solution we manually upgraded the client test

suite and used it as the base case. The overhead of using the layer instead of upgrading the client is small. In cases that involve the encapsulation of fields with accessor methods, the runtime overhead is negative because the adapted version directly references the fields, whereas the refactored version uses the accessor methods.

6.7.2 Case Studies

We took three core Eclipse libraries and examined the refactorings executed by the Eclipse developers. Each row in Table 6.3 shows the versions of the library and client. The client `workbench.texteditor.tests` is an older version of an Eclipse test suite comprising 106 tests. The third row shows an example of an Eclipse library, `core.refactoring` breaking an Eclipse UI client, `ui.refactoring`.

Table 6.4 displays the effectiveness and efficiency of the compatibility layer generated by ReBA. After using the compatibility layer generated by ReBA, all but one tests passed for the first two case studies. However, the single failing test is a test that was failing even in the scenario where library and tests in Eclipse are compatible (both have the same version number).

In the third study, since Eclipse CVS repository does not contain any tests for the client, we could not run any automated test suite before/after applying the compatibility layer. However, since the `ui.refactoring` is a graphical client, we exercised it manually after applying the ReBA-generated compatibility layer. All usage scenarios reveal that the client is working properly.

To answer the efficiency question, we examined the compatibility layer and found that it contains only the classes that are truly needed by the client. To measure the memory and running overhead, we manually upgraded the source code of the client to make the client compilable; we used this upgraded client as the base for comparison with the compatibility layer. The memory overhead is much smaller in the case studies because the compatibility layer is very small compared to the large number of classes in each library.

All experiments were performed on a laptop with Intel Pentium M, 1.6GHz processor, and 512MB of RAM. The analysis done by WALA is scalable. WALA takes 2 minutes to create the points-to graph for `jdt.ui`, a relatively large library of 461 KLOC.

Table 6.3: Eclipse plugins used as case studies.

Library	LibrarySize[LOC]	client
`workbench.texteditor` v20060905	16842	`workbench.texteditor.tests` v20060829 (106 tests)
`jface.text` v20060926	31551	`workbench.texteditor.tests` v20060829 (106 tests)
`ltk.core.refactoring` v20060228	8121	`ltk.ui.refactoring` v20060131

Table 6.4: Effectiveness and efficiency of ReBA using Eclipse plugins as case studies.

Study	API-Breaking Changes	CompileErrors		RuntimeFailures		MemoryUsage[B]		Memory Overhead	RunningTime[s]		Running Overhead
		Before	After	Before	After	Before	After		Before	After	
#1	1 RenameClass	29	0	25	1	140003416	140004056	$457*10^{-6}$%	14.746	14.748	0.009%
#2	1 DeleteClass	6	0	18	1	140044056	140044072	$11*10^{-6}$%	14.67	14.40	0.09%
#3	4 DelClass, 4 DelMethod, 1 RenaMethod 2 ChangeMethSig, 1 MoveClass, 1 DelField	27	0	12	0	17589288	17589402	$648*10^{-6}$%	8.452	8.453	0.01%

6.8 Summary

Managing software evolution is complex, and no technique will be a silver bullet. In the long run, source code must evolve to be compatible with new versions of libraries. In the short run, however, it is valuable to allow new versions of libraries to replace their old versions without changing the clients that use them. Our method of binary compatibility, embodied in the ReBA system, allows a library to simultaneously support the old and the new APIs. This allows it to be used in a system with both updated and non-updated clients. The producer of the library will make an adapted version that supports both versions of the API, but which has a lot of redundancy. The consumer of the library then makes a version of the adapted-library that eliminates the redundancy by specializing it to the client that is using it. The result, as shown by our evaluation, is a system that is both effective and efficient.

Compared to other binary compatibility techniques, our solution is easy to apply and does not require modifying bytecodes. It should be considered by library producers who are worried about the cost of library evolution on their customers.

Chapter 7

Related Work

This dissertation research spans several areas of software engineering. Here we provide an overview of related work grouped by the following areas: studies of API evolution, refactoring, automated detection of refactorings, software merging, and binary component adaptation.

7.1 API Evolution

To the best of our knowledge, ours is the first quantitative and qualitative study about the kind of API changes that occur in components.

Bansiya [Ban99] and Mattson [MB99] used metrics to assess the stability of frameworks. Their metrics can only detect the effect of changes in the framework and not the exact type of change (e.g. they observed that method argument types have been changed between subsequent versions whereas we observe whether they changed because of adding/removing of parameters or because of changing the argument types). However, their empirical analysis is consistent with our findings: most of the changes occur as responsibility is shifted around classes (e.g. methods or fields moved around), collaboration protocol changes (e.g. different method signatures) and new classes are added.

Mattson and Bosch [MB98] identified four evolution categories in frameworks: internal reorganization, changing functionality, extending functionality and reducing functionality. Our findings confirm all four of the evolutions they have been describing.

Kim et al. [KWJ06] analyze the signature-change patterns when a software system evolves. They consider seven open-source software systems written in C. They conclude that the most common type of signature change is parameter addition, followed by complex type change and parameter deletion.

Tool support for upgrading applications when the APIs of their components change has been a long time interest. [CN96, KH98, RH02] discuss different annotations within the component's source code that

can be used by tools to upgrade applications. However, writing such annotations is cumbersome. Balaban et al. [BTF05] aim to automatically replace obsolete library classes with their newer counterparts. Component developers have to provide mappings between legacy classes and their replacements. A more appealing approach would be if tools could generate this information, therefore reducing the overhead on the component producers.

As an alternative to refactorings, Steyaert et al. [SLMD96] introduce the notion of Reuse Contracts to guarantee structural and behavioral compatibility between frameworks and instantiations. On the same line, Tourwé and Mens [TM03] introduce metapatterns and their associated transformations to document the framework changes. Because of the rich semantics carried in such documentation, automated support for application migration can be possible. We agree that refactoring alone cannot solve all the migration problems. However, automated refactoring is supported by most recent IDEs. We showed that refactorings can effectively describe over 80% of the breaking API changes that actually occur during component evolution.

7.1.1 Refactoring

Programmers have been cleaning up their code for decades, but the term *refactoring* was coined much later [OJ90]. Opdyke [Opd92] wrote the first catalog of refactorings, while Roberts and Brant [RBJ97, Rob99] were the first to implement a refactoring engine. The refactoring field gained much popularity with the catalog of refactorings written by Fowler et al. [FBB+99]. Soon after this, IDEs began to incorporate refactoring engines. Tokuda and Batory [TB01] describe how large architectural changes in two frameworks can be achieved as a sequence of small refactorings. They estimate that automated refactorings are 10 times quicker to perform than manual ones.

More recent research on refactoring focuses on the analyses for automating type-related refactorings. Tip et al. [TKB03] use type constraints to support an analysis for refactorings that introduce type generalization. Donovan et al. [DKTE04] use a pointer analysis and a set-constraint-based analysis to support refactorings that replace the instantiation of raw classes with generic classes. Dincklage and Diwan [vDD04] use various heuristics to convert from non-generic classes to generic classes. Balaban et al. [BTF05] propose refactorings that automatically replace obsolete library classes with their newer counterparts. Component developers have to provide mappings between legacy classes and their replacements, and an analysis based on type constraints determines where the replacement can be done. Thomas [Tho05] points out that refac-

torings in the components result into integration problems and advocates the need for languages that would allow developers to create customizable refactorings that can upgrade applications.

7.2 Detection of Refactorings

Tool support for detecting and classifying structural evolution has spawned mostly from the reengineering community. Detection of class splitting and merging was the main target of the current tools. Demeyer et al. [DDN00] use a set of object-oriented change metrics and heuristics to detect refactorings that will serve as markers for the reverse engineer. Antoniol et al. [APM04] use a technique inspired from the Information Retrieval to detect discontinuities in classes (e.g., a class was replaced with another one, a class was split into two, or two classes merge into one). Based on Vector Space cosine similarity, they compare the class identifiers found in two subsequent releases. Therefore, if a class, say `Resolver`, was present in version n but disappears in version $n + 1$ and a new class `SimpleResolver` appears in version $n + 1$, they conclude that a class replacement happened. Godfrey and Zou [GZ05] are the closest to the way how we envision detecting structural changes. They implemented a tool that can detect for procedural code some refactorings like renaming, move method, split, and merge. Whereas we start from shingles analysis, they employ origin analysis along with a more expensive analysis on call graphs to detect and classify these changes. Rysselberghe and Demeyer [RD03] use a clone finding tool (Duploc) to detect methods that were moved across the classes. Weissgerber and Diehl [WD06] analyze subsequent versions of a component in configuration management repositories to detect refactorings and later use code-clone detection to refine the results. Kim et al. [KPEJW05] propose an algorithm based on heuristics to detect rename method refactorings between two versions of a software. They use eight similarity factors to detect these refactorings. Xing and Stroulia [XS06] detect refactorings at a design level from UML diagrams using the structural changes between the two versions of the diagrams.

Existing work on automatic detection of refactorings addresses some of the needs of reverse engineers who must understand at a high level how and why components evolved. For this reason, most of the current work focuses on detecting merging and splitting of classes. However, in order to automatically upgrade component-based applications we need to know the changes to the API. Our work complements existing work because we must look also for lower level refactorings that affect the signatures of methods. We also address the limitations of existing work with respect to renamings and noise introduced by multiple

refactorings on the same entity or the noise introduced by the deprecate-replace-remove cycle in the open-world components.

7.2.1 Shingles Encoding

Clone detection based on Shingles encoding is a research interest in other fields like internet content management and file storage. Ramaswamy et al. [RILD04] worked on automatic detection of duplicated fragments in dynamically generated web pages. Dynamic web pages cannot be cached, but performance can be improved by caching fragments of web pages. They used Shingles encoding to detect fragments of web pages that did not change. Manber [Man93] and Kulkarni et al. [KDLT04] employ shingles-based algorithms to detect redundancy in the file system. They propose more efficient storage after eliminating redundancy. Li et al. [LLMZ04] use shingles to detect clones of text in the source code of operating systems. Their tool, CPMiner, further analyzes the clones to detect bugs due to negligent copy and paste.

7.3 Software Merging

According to Mens [Men02], software merging techniques can be distinguished based on how software artifacts are represented. *Text-based* merge tools consider software artifacts merely as text (or binary) files. In RCS and CVS [Mor96], lines of text are taken as indivisible units. Despite its popularity, this approach cannot handle well two parallel modifications to the same line. Only one of the two modifications can be selected, but they cannot be combined. *Syntactical* merging is more powerful than textual merging because it takes the syntax of software artifacts into account. Unimportant conflicts such as code comment or line breaks can be ignored by syntactic merger. Some syntactic merge tools focus on parse-trees or abstract syntax tree [Ask94, Gra92a, Yan94]. Other are based on graphs [Men99, RW98]. However, they cannot detect conflicts when the merged program is syntactically correct but semantically invalid. To deal with this, *semantic-based* merge algorithms were developed. In Wesfetchtel's context-sensitive merge tool [Wes91], an AST is augmented by the bindings of identifiers to their declarations. More advanced semantic-based merge algorithms [Ber94, HS77, YHR92] detect behavioral conflicts using dependency graphs, program slicing, and denotational semantics.

Operation-based Merging. The operation-based approach has been used in software merging [Edw97, LCDK89, LvO92, Men99, SS04]. It is a particular flavor of semantic-based merging that models changes

between versions as explicit operations or transformations. Operation-based merge approach can improve conflict detection and allows better conflict solving [Men02]. Lippe *et al.* [LvO92] describes a theoretical framework for conflict detection with respect to general transformations. No concrete application for refactorings was presented. Edwards' operation-based framework detects and resolves semantic conflicts from application-supplied semantics of operations [Edw97]. GINA [BG93] used a *redo* mechanism to apply one developer's changes to the other developer's version. The approach has problems with long command histories and finer granularity of operations. The departure point of MolhadoRef from existing approaches is its ability to handle the merging of changes that involve both *refactoring* and *textual editing*.

Similar to MolhadoRef, Ekman and Asklund [EA04] present a refactoring-aware versioning system. Their approach keeps the program elements and their IDs in volatile memory, thus allowing for a short-lived history of refactored program entities. In our approach, program elements and their IDs are modeled in the SCM and stored throughout the lifecycle of the project, allowing for a global history tracking of refactored entities. Also, their system does not support merging.

As described, fine-grained and ID-based versioning have been proposed before by others. However, the novelty of this work is the combination of semantic-based, fine-grained, ID-based SCM to handle refactorings and high-level edit operations. To the best of our knowledge, we are presenting the first algorithm to merge refactorings and edits. The algorithm is implemented and the first experiences are demonstrated.

7.4 Binary Component Adaptation

There are several previous solutions [PA91, FCDR95, KH98, WSM06, SR07] that aim to restore the binary compatibility. Purtilo and Atlee [PA91] present a language, called Nimble, that allows one to specify how the parameters of a function call from an old module can be adapted to match the new signature of the function. Nimble handles a rich set of changes in function signatures: reordering of parameters, coercing primitive types into other primitive types, or it can project a subset of the arguments. Nimble works for procedural languages, whereas ReBA is an approach for OO languages. The function signature changes in Nimble are similar with the ChangeMethodSignature refactorings. In addition, ReBA supports other types of API changes.

Keller and Hölzle [KH98] present BCA, a tool that automatically rewrites the bytecodes of Java clients using information that a user supplies in delta files. BCA compiles the delta files and automatically inserts

the bytecode patches into bytecodes of clients before loading the client classes. Although BCA can handle a large range of API changes, it requires that developers use the BCA compiler and class loader. For security and legal reasons, developers are reluctant to use solutions that require changing the bytecodes.

The Adapter/Wrapper [GHJV95] pattern adapts a class's interface to the interface that a client expects. The adapter/wrapper object delegates to the adaptee/wrapped object. This solution works fine when the client code can be changed to use the wrapper wherever it used the original class. However, the adapter pattern has problems when restoring the binary compatibility. Because the wrapper and the wrapped object need to have different names, their identity and type information is different. Clients could potentially end up with objects of the wrapper as well as wrapped objects, depending on whether the client instantiates the wrapper directly, or it gets an indirect instance from the library. Comparing the wrapper and the wrapped through == or `instanceof` would not work.

Warth et al. [WSM06] present a technique based on expanders. Expanders are wrappers automatically generated from annotations provided by the user. However, expanders are only available in code that explicitly imports them, thus this technique requires that client code knows a priori about the expanders. Being based on the Adapter pattern, expanders suffer from the loss of identity and type information.

Savga and Rudolf [SR07] overcome the above limitation by ensuring that (i) the wrapper has the same type information as the one expected by the client and (ii) the library itself would always return only instances of the wrapper objects. To restore the old API of the library they execute *comebacks* which are program transformations that revert all the API refactorings in the library. This enables old clients to work, but not new clients. In contrast, our solution does not modify the library, thus new clients continue to work. Their solution, like ours, preserves the edits in the library. However, their solution generates wrappers for all public classes in the library, whereas our solution generates adapted-classes only for the classes that have changed through refactoring. In addition, our solution handles the deletion of APIs.

We solve the object identity problem in a different way. Rather than creating a wrapper and a wrapped object, we combine the wrapper (the old interface) and the wrapped object (the new interface) into one single class, the adapted-class. In addition, the points-to analysis ensures that the client and the compatibility layer always instantiate the same class, namely the adapted-class, thus preserving object identity and type information.

There is large body of work in the area of bridging components using architectural connectors [AG94].

Architectural connectors can be used to adapt interface mismatches. Although architectural connectors can adapt other types of changes besides syntactic interface change (e.g., behavioral, protocol), this approach requires that library developers write formal specifications. From these specifications, tools can automatically synthesize adapters, for example as in [YS97, AINT07]. ReBA does not make any behavioral guarantees since it handles edits as well, although it ensures that the behavior of the client is not changed with respect to API refactorings in the library. However, ReBA is practical because it does not require that library developers write specifications.

Lastly, most previous solutions (except [HD05, SR07]) demand that library developers write annotations/mappings/specifications that are going to be used by the upgrading tools. Library developers are usually reluctant to write such annotations. ReBA harvests the refactoring information from the library's refactoring engine, while liberating the library developers from the burden of manually writing annotations.

Chapter 8

Conclusion

API changes have an impact on applications. One might argue that component developers should maintain old versions of the component so that applications built on those versions continue to run. However, this results in version proliferation and high maintenance costs for the producer. In practice, it is application developers who adapt to the changes in the library.

Evolution of software components is a complex and large topic. No single tool could handle all its aspects. For this dissertation we have conducted several investigations. The first one characterizes changes in real world components. The next ones build and evaluate a tool-set that automates upgrades.

We looked at five widely used components and studied what changed between two major releases. Then we analyzed those changes in detail and found out that in the five case studies, respectively 84%, 81%, 90%, 97%, and 94% of the API breaking changes are structural, behavior-preserving transformations (refactorings). This confirms that refactoring plays an important role in the evolution of components.

If component refactorings are automatically detected, they can be automatically incorporated into applications. A tool like Eclipse can replay these refactorings. The replay is done at the application site where both the component and the application reside in the same workspace. In this case, the refactoring engine finds and correctly updates all the references to the refactored entities, thus upgrading the application to the new API of the component.

One way to get sequences of refactorings is to record them in the IDE. But programmers might refactor manually, and old versions of components predate refactoring tools. So, it is important to detect refactorings that occurred between two versions of a component. For the purpose of detecting refactorings in real-world components, syntactic analyses are too unreliable, and semantic analyses are too slow. Combining syntactic and semantic analyses can give good results. By combining Shingles encoding with traditional semantic analyses, and by iterating the analyses until a fixed point was discovered, our tool RefactoringCrawler detects over 85% of the refactorings while producing less than 10% false positives.

Incorporating component refactorings into an application is a special case of merging refactorings and edits from component and application. We developed a novel SCM system, MolhadoRef, that is aware of program entities and the refactoring operations that change them. MolhadoRef uses the operation-based approach to record (or detect) and replay changes. By intelligently treating the dependences between different operations, it merges edit and refactoring operations effectively. In addition, because MolhadoRef is aware of the semantics of change operations, a successful merge does not produce compile or runtime errors.

As an alternative to changing application source code to accommodate the changes in components, we developed ReBA, a tool that automatically generates a compatibility layer between component and application. This layer is automatically generated from the trace of component refactorings and enables old applications to run along with the refactored version of the component, without requiring any changes to the application.

8.1 Making a Difference

This dissertation is about solving practical problems; our goal is to influence and improve the lives of software developers. The ultimate test for impact is whether industry adopts and further perfects these techniques and prototypes. We believe that research prototypes are more likely to be tried and used by industry (open-source and commercial) when they are conveniently incorporated in integrated development environments (IDEs) such as Eclipse. This is why our toolkit is integrated entirely with the Eclipe IDE. Thus, we expect to have a large customer base. We expect this will only be the beginning. The user base will grow and we will learn a lot from users.

Some of our tools have already been downloaded by several hundred developers, others by millions of developers (the record-and-replay refactoring engine in Eclipse). We have received good reports from researchers who use our tools at four different institutions (North Carolina State University, University of Montreal, Portland State University, Iowa State University). One tool, RefactoringCrawler, has been further extended [TDX07] at North Carolina State University. RefactoringCrawler is used in industry (CuramSoftware) to validate that the release notes contain all API changes in a given release.

In the long run, our solutions will only be picked up by the masses if they are introduced by the IDEs. We maintain close relationships with companies that develop IDEs. Most notably, our collaboration with IBM Zurich led to the integration of record-and-replay feature in the official release of Eclipse. Both Microsoft

and IBM have shown a lot of interest for integrating our refactoring-aware software merging algorithm into the IDEs they develop. In addition, IBM is particularly interested in ReBA, our adaptation tool; in fact the idea of generating adapters between new versions of components and old versions of applications came out during a summer internship with the developers of Eclipse IDE. We plan to nurture our current relationships and assist IDE producers in turning our solutions into industrial-strength products.

8.2 Future Work

8.2.1 Toolkit Evaluation

Although we evaluated individual tools, we plan to evaluate how the whole toolkit impacts the practice of component-based software development. The goal of future evaluation is two-fold: (i) we want to asses the impact of our solutions upon application developers and (ii) component developers. More specifically, we want to answer these questions:

First, how much of the effort spent on upgrading component-based applications can be saved? We plan to evaluate how the whole migration toolkit eases the task of the application developer when upgrading to a new version of the component. We plan to compare the productivity of a control group against that of a group that uses our toolkit.

Second, how would component developers evolve the component's design when, because of this powerful upgrading technology, they do not have to worry about backwards compatibility? We will observe what kind of changes component designers make when they do not have to worry about breaking the compatibility with existing applications. We hypothesize that the availability of such powerful upgrading tools will encourage component developers to further refactor their designs to be easier to understand and reuse.

8.2.2 Algebra of Program Transformations

One of the problems with the current implementation of refactoring tools is that sometimes the analysis they perform is too ad hoc. This makes it hard to analytically derive the safety checks that an upgrading tool needs to perform. A possible approach to put refactorings on a more rigorous track is to view them as algebraic functions, similar to the terms used in architectural metaprogramming literature [Bat07]. We plan to revisit all transformations that we encountered in real world components and express them as algebraic

functions.

Developing an algebra of program transformations helps us to mechanize the safety checks that an upgrading tool performs. Currently, we hardcoded such checks in conflict and dependency matrices. Once the API changes are defined as algebraic functions, an upgrading tool could check whether their composition is valid.

8.2.3 Checking The Behavior of Upgraded Applications

With respect to the API changes caused by API refactorings, our tools ensure that the upgraded application has the same behavior before and after the upgrade. However, besides structural changes, new versions of components might contain behavioral changes. Upgrading an application such that it takes into account the behavioral changes is a much more ambitious topic. With respect to behavioral changes, ensuring that an upgraded application runs correctly is in general undecidable or infeasible. This would require that developers write full formal specifications of each change they make. In the absence of such complete specifications, our upgrading tools cannot guarantee that the upgrade is successful, thus requiring that application developers run a regression test suite after the upgrade.

In the future, we plan to complement our approach with a lightweight approach to infer specifications in components. We plan to use a tool like Daikon [Ern00] to dynamically discover program invariants in components. Any discrepancy in the invariants before and after the upgrade is brought into the attention of the human who can decide how to change the application to make up for the change in behavior. We believe that programmers enjoy working on behavioral upgrading tasks that are intellectually stimulating; as for the tedious structural upgrading tasks, it is best if they are done by computers. This dissertation shows that it is both practical and safe to automate the structural upgrades.

8.2.4 Handling Other Program Transformations

Other program transformations can be dealt with similarly with how this dissertation deals with refactorings: detecting, merging, and adapting to them. Next, we will work on one of the main trends of computing: the adoption of parallel computing to make use of today's multi-core architectures. There is an urgent need to parallelize existing programs. Traditionally, this is either done by hand or is entirely done by computers. But there is third way. Refactoring tools have shown that the human can do the creative part (making

the decision of what to parallelize), while the machine does the bookkeeping (checking the safety of the transformation and carrying it out). We plan to apply the techniques developed for this dissertation to the problem of retrofitting parallelism into existing programs.

8.3 Final Remarks

Traditionally, the programming task has been thought of as turning requirements or specifications into an executable program. In the 21st century, most programmers are transforming old versions of a program into new versions. Only a fraction of these changes are refactorings, but all of them are program transformations.

Over time, the percentage of program transformations that are automated will increase. Our work on refactoring is a model of what will need to be done for any kind of program transformation. Although the interactive tools in the IDE can record the transformations, there will still be a need for detecting transformations. There will also be a need to merge sequences of program transformations. In a large system, some parts will not be transformable, and some sort of adapter will need to be produced. Although the solution details might change, the problems that we have worked on in this dissertation will continue to be important.

References

[AG94] Robert Allen and David Garlan. Formalizing architectural connection. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 71–80, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[AINT07] Marco Autili, Paola Inverardi, Alfredo Navarra, and Massimo Tivoli. Synthesis: A tool for automatically assembling correct and distributed component-based systems. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 784–787, Washington, DC, USA, 2007. IEEE Computer Society.

[APM04] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo. An automatic approach to identify class evolution discontinuities. In *IWPSE'04: Proceedings of International Workshop on Principles of Software Evolution*, pages 31–40, 2004.

[Ask94] Ulf Asklund. Identifying conflicts during structural merge. In *Proceedings of Nordic Workshop on Programming Environment Research*, pages 231–242, 1994.

[Ban99] J. Bansiya. *Object-Oriented Application Frameworks: Problems and Perspectives*, chapter Evaluating application framework architecture structural and functional stability. Wiley, 1999.

[Bat07] Don S. Batory. Program refactoring, program synthesis, and model-driven development. In *CC'07: Proceedings of 16th International Conference on Compiler Construction, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007*, volume 4420 of *Lecture Notes in Computer Science*, pages 156–171. Springer, 2007.

[Bec97] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.

[Ber94] Valdis Berzins. Software merge: semantics of combining changes to programs. *ACM Trans. Program. Lang. Syst.*, 16(6):1875–1903, 1994.

[BG93] Thomas Berlage and Andreas Genau. A framework for shared applications with a replicated architecture. In *Proceedings of the Symposium on User Interface Software and Technology*, pages 249–257. ACM Press, 1993.

[Bro97] Andrei Broder. On the resemblance and containment of documents. In *SEQUENCES '97: Proceedings of Compression and Complexity of Sequences*, pages 21–29, 1997.

[BTF05] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. In *OOPSLA '05: Proceedings of Object-oriented programming, systems, languages, and applications*, pages 265–279, New York, NY, USA, 2005. ACM Press.

[CHK+01] Ned Chapin, Joanne E. Hale, Khaled Md. Khan, Juan F. Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance*, 13(1):3–30, 2001.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*, pages 549–552. The MIT Press and McGraw-Hill Book Company, 2001.

[CN96] Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *ICSM'96: Proceedings of International Conference on Software Maintenance*, pages 359–368. IEEE Computer Society, 1996.

[Dan] Danny Dig. Informal interviews with Eclipse component developers, conducted during one summer internship at IBM Research Zurich, 2004.

[DCMJ06] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automatic detection of refactorings in evolving components. In *ECOOP'06: Proceedings of European Conference on Object-Oriented Programming*, pages 404–428. Springer Verlag, 2006.

[DDN00] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *OOPSLA'00: Proceedings of Object oriented programming, systems, languages, and applications*, pages 166–177, 2000.

[DDN03] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2003.

[DJ05] Danny Dig and Ralph Johnson. The role of refactorings in API evolution. In *ICSM'05: Proceedings of International Conference on Software Maintenance*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

[DJ06] Danny Dig and Ralph Johnson. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution*, 18(2):87–103, 2006.

[DKTE04] Alan Donovan, Adam Kiezun, Matthew S. Tschantz, and Michael D. Ernst. Converting Java programs to use generic libraries. In *OOPSLA '04: Proceedings of Object-oriented programming, systems, languages, and applications*, volume 39, pages 15–34, New York, NY, USA, October 2004. ACM Press.

[DMJN06] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien Nguyen. Refactoring-aware configuration management system for object-oriented programs. Technical Report UIUCDCS-R-2006-2770, UIUC, September 2006.

[DMJN07a] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen. Refactoring-aware Configuration Management for Object-Oriented Programs. In *Proceedings of International Conference on Software Engineering*, pages 427–436. IEEE Computer Society, 2007.

[DMJN07b] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen. Refactoring-aware software merging in the presence of object-oriented refactorings. *Accepted to IEEE Transactions of Software Engineering*, 2007.

[DNJ06] Danny Dig, Tien Nguyen, and Ralph Johnson. Refactoring-aware software configuration management. Technical Report UIUCDCS-R-2006-2710, UIUC, April 2006.

[DNJM07] Danny Dig, Stas Negara, Ralph Johnson, and Vibhu Mohindra. Reba: Refactoring-aware binary adaption of evolving libraries. Technical Report UIUCDCS-R-2007-2899, UIUC, September 2007.

[DRG+05] Serge Demeyer, Filip Van Rysselberghe, Tudor Gîrba, Jacek Ratzinger, Radu Marinescu, Tom Mens, Bart Du Bois, Dirk Janssens, Stéphane Ducasse, Michele Lanza, Matthias Rieger, Harald Gall, and Mohammad El-Ramly. The LAN-simulation: A Refactoring Teaching Example. In *Proceedings of International Workshop on Principles of Software Evolution*, pages 123–134. IEEE Computer Society, 2005.

[EA04] Torbjörn Ekman and Ulf Asklund. Refactoring-aware versioning in eclipse. *Electr. Notes Theor. Comput. Sci.*, 107:57–69, 2004.

[Ecla] Eclipse Foundation. http://eclipse.org.

[Eclb] What's new in Eclipse 3.2 (JDT). http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.jdt.doc.user/whatsNew/jdt_whatsnew.html.

[Edw97] W.K. Edwards. Flexible Conflict Detection and Management in Collaborative Applications. In *Proceedings of the Symposium on User Interface Software and Technology*, pages 139–148. ACM Press, 1997.

[Ern00] Michael Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington, 2000.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Adison-Wesley, 1999.

[FCDR95] Ira R. Forman, Michael H. Conner, Scott H. Danforth, and Larry K. Raper. Release-to-release binary compatibility in som. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 426–438, New York, NY, USA, 1995. ACM Press.

[GDL04] Tudor Gîrba, Stéphane Ducasse, and Michele Lanza. Yesterday"s Weather: Guiding Early Reverse Engineering Efforts by Summarizing the Evolution of Changes. In *Proceedings of International Conference on Software Maintenance*, pages 40–49. IEEE Computer Society, 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gra92a] Judith E. Grass. Cdiff: A Syntax Directed Differencer for C++ Programs. In *Usenix C++ Conference*, pages 181–194, 1992.

[Gra92b] Justin O. Graver. The evolution of an object-orietented compiler framework. *Softw., Pract. Exper.*, 22(7):519–535, 1992.

[GW05] Carsten Gorg and Peter Weisgerber. Detecting and visualizing refactorings from software archives. In *IWPC'05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 205–214, Washington, DC, USA, 2005. IEEE Computer Society.

[GZ05] Michael W. Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[HD05] Johannes Henkel and Amer Diwan. CatchUp!: Capturing and replaying refactorings to support API evolution. In *ICSE'05: Proceedings of International Conference on Software Engineering*, pages 274–283, 2005.

[HS77] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, 1977.

[Jav] JavaRefactor refactoring engine for JEdit IDE. http://plugins.jedit.org/plugins/?JavaRefactor.

[JHoa] JHotDraw Framework. http://www.jhotdraw.org.

[JHob] JHotDraw Applications. http://www.jhotdraw.org/survey/applications.html.

[Joh92] Ralph E. Johnson. Documenting frameworks using patterns. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 63–76, New York, NY, USA, 1992. ACM Press.

[KDLT04] Purushottam Kulkarni, Fred Douglis, Jason D. LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *USENIX Annual Technical Conference, General Track*, pages 59–72, 2004.

[KH98] Ralph Keller and Urs Hölzle. Binary component adaptation. *Lecture Notes in Computer Science*, 1445:307–329, 1998.

[KPEJW05] Sunghun Kim, Kai Pan, and Jr. E. James Whitehead. When functions change their names: Automatic detection of origin relationships. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 143–152, Washington, DC, USA, 2005. IEEE Computer Society.

[KWJ06] Sunghun Kim, E. James Whitehead, and . Jennifer Bevan Jr. Properties of signature change patterns. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 4–13, Washington, DC, USA, 2006. IEEE Computer Society.

[Lai99] M. Laitinen. *Object-Oriented Application Frameworks: Problems and Perspectives*, chapter Framework maintenance: Vendor viewpoint. Wiley, 1999.

[LCDK89] A. Lie, R. Conradi, T. M. Didriksen, and E.-A. Karlsson. Change oriented versioning in a software engineering database. In *Proceedings of the 2nd Workshop on Software configuration management*, pages 56–65. ACM Press, 1989.

[LLMZ04] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI'04: Proceedings of the Sixth Symposium on Operating System Design and Implementation*, pages 289–302, 2004.

[Log] Apache Foundation log4J Library. http://logging.apache.org/log4j.

[LS80] B. P. Lientz and E. B. Swanson. *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations.* Adison-Wesley, 1980.

[LvO92] Ernst Lippe and Norbert van Oosterom. Operation-based merging. In *Proceedings of Symposium on Software Development Environments*, pages 78–87. ACM Press, 1992.

[LY01] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification, Second Edition*, pages 102–103. Prentice Hall, 2001.

[Man93] Udi Manber. Finding similar files in a large file system. Technical Report 93-33, University of Arizona, 1993.

[MB98] M. Mattson and J. Bosch. Frameworks as components: a classification of framework evolution. In *NWPER'98: Nordic Workshop on Programming Environment Research*, pages 16–74, 1998.

[MB99] M. Mattson and J. Bosch. Three evaluation methods for object-oriented frameworks evolution - application, assessment and comparison. Technical Report 1999:20, Department of Computer Science, University of Karlskrona/Ronneby, Sweden, 1999.

[MBF99] Michael Mattsson, Jan Bosch, and Mohamed E. Fayad. Framework integration problems, causes, solutions. *Commun. ACM*, 42(10):80–87, 1999.

[Men99] Tom Mens. *A Formal Foundation for Object-Oriented Software Evolution*. PhD thesis, Vrije Universiteit Brussel, 1999.

[Men02] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[Mey05] Berthrand Meyer. *Design by Contract*. Prentice Hall, 2005.

[MM85] Webb Miller and Eugene W. Myers. A file comparison program. *Softw., Pract. Exper.*, 15(11):1025–1040, 1985.

[Mol] Molhadoref's web page. https://netfiles.uiuc.edu/dig/MolhadoRef.

[Mor96] Tom Morse. CVS. *Linux Journal*, 1996(21es):3, 1996.

[Net] NetBeans ide and refactoring engine. http://netbeans.org.

[NMBT05] Tien Nhut Nguyen, Ethan V. Munson, John Tang Boyland, and Cheng Thao. An Infrastructure for Development of Object-oriented, Multi-level Configuration Management Services. In *Proceedings of International Conference on Software Engineering*, pages 215–224. ACM Press, 2005.

[OJ90] Bill Opdyke and Ralph Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *SOOPPA'90: Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications*, 1990.

[Opd92] Bill Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[PA91] James M. Purtilo and Joanne M. Atlee. Module reuse by interface adaptation. *Softw. Pract. Exper.*, 21(6):539–556, 1991.

[Rab81] Michael O. Rabin. Fingerprinting by random polynomials. Technical Report 15-81, Harvard University, 1981.

[RBJ97] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *TAPOS*, 3(4):253–263, 1997.

[RD03] Filip Van Rysselberghe and Serge Demeyer. Reconstruction of successful software evolution using clone detection. In *IWPSE'03: Proceedings of 6th International Workshop on Principles of Software Evolution*, pages 126–130, 2003.

[Ref] Refactoringcrawler's web page. https://netfiles.uiuc.edu/dig/RefactoringCrawler .

[RH02] Stefan Roock and Andreas Havenstein. Refactoring tags for automatic refactoring of framework dependent applications. In *XP'02: Proceedings of Extreme Programming Conference*, pages 182–185, 2002.

[RILD04] Lakshmish Ramaswamy, Arun Iyengar, Ling Liu, and Fred Douglis. Automatic detection of fragments in dynamically generated web pages. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 443–454, New York, NY, USA, 2004. ACM Press.

[Riv] James Des Riviers. Evolving Java-based APIs. http://www.eclipse.org/eclipse/development/java-api-evolution.html.

[Rob99] Don Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[RW98] Jungkyu Rho and Chisu Wu. An efficient version model of software diagrams. In *Proceedings of the Fifth Asia Pacific Software Engineering Conference*, pages 236–243. IEEE Computer Society, 1998.

[SLMD96] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse contracts: managing the evolution of reusable assets. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 268–285, New York, NY, USA, 1996. ACM Press.

[SR07] Ilie Savga and Michael Rudolf. Refactoring-based support for binary compatibility in evolving frameworks. In *GPCE'07: International Conference on Generative Programming and Component Engineering*, page To Appear, 2007.

[SS04] Haifeng Shen and Chengzheng Sun. A complete textual merging algorithm for software configuration management systems. In *Proceedings of the 28th Annual International Computer Software and Applications Conference*, pages 293–298. IEEE Computer Society, 2004.

[Str] Apache Foundation Struts Framework. http://struts.apache.org.

[TB01] Lance Tokuda and Don Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, 8(1):89–120, January 2001.

[TDX07] Kunal Taneja, Danny Dig, and Tao Xie. Automated detection of api refactorings in libraries. In *ASE'07: Proceedings of International Conference on Automated Software Engineering, (To Appear)*, 2007.

[Tho05] Dave Thomas. Refactoring as meta programming? *Journal of Object Technology*, 4(1):7–11, January-February 2005.

[Tip07] Frank Tip. Refactoring using type constraints. In *SAS'07: Proceedings of 14th International Symposium on Static Analysis*, volume 4634 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2007.

[TKB03] Frank Tip, Adam Kiezun, and Dirk Bauemer. Refactoring for generalization using type constraints. In *OOPSLA '03: Proceedings of Object-oriented programing, systems, languages, and applications*, pages 13–26, New York, NY, USA, November 2003. ACM Press.

[TM03] Tom Tourwé and Tom Mens. Automated support for framework-based software evolution. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 148, Washington, DC, USA, 2003. IEEE Computer Society.

[vDD04] Daniel von Dincklage and Amer Diwan. Converting Java classes to use generics. In *OOPSLA '04: Proceedings of Object-oriented programming, systems, languages, and applications*, pages 1–14. ACM Press, 2004.

[WAL] WALA Static Analysis Library. http://wala.sourceforge.net/.

[WD06] Peter Weissgerber and Stephan Diehl. Identifying refactorings from source-code changes. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society.

[Wei80] William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 83–94, New York, NY, USA, 1980. ACM Press.

[Wes91] Bernhard Westfetchtel. Structure-oriented Merging of Revisions of Software Documents. In *Proceedings of the 3rd International Workshop on Software Configuration Management*, pages 68–79. Springer Verlag, 1991.

[WSM06] Alessandro Warth, Milan Stanojević, and Todd Millstein. Statically scoped object adaptation with expanders. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 37–56, New York, NY, USA, 2006. ACM Press.

[XS06] Zhenchang Xing and Eleni Stroulia. Refactoring detection based on umldiff change-facts queries. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 263–274, Washington, DC, USA, 2006. IEEE Computer Society.

[Yan94] Wuu Yang. How to merge program texts. *The Journal of Systems and Software*, 27(2), November 1994.

[YHR92] Wuu Yang, Susan Horwitz, and Thomas Reps. A program integration algorithm that accommodates semantics-preserving transformations. *ACM Trans. Softw. Eng. Methodol.*, 1(3):310–354, 1992.

[YS97] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997.

Author's Biography

Daniel Dig was born on December 16th, 1977, in Transylvania, a province of Romania. He attended the Polytechnics University of Timisoara where he got a B.S. degree in Computer Science in 2001. During the course of his studies, he was awarded the *Performance Scholarship* by Romanian Ministry of Education for scoring perfect grades for at least three consecutive semesters. During 1997 - 2001 he was also awarded the Deans Scholarship at the Polytechnics University for being among the top 3% students in his generation. For his thesis project he developed and released JavaRefactor, the first open-source refactoring engine for Java, getting over 17,000 downloads in the first 6 months after the release.

Dr. Dig immediately enrolled in the Master's program at the Polytechnics University of Timisoara. In 2002 he earned the M.S. in Computer Science. His dissertation was entitled *Refactoring to Patterns: An Automated Approach*.

Dr. Dig enrolled in the Ph.D. program at the University of Illinois in 2002 under the supervision of Dr. Ralph Johnson. His doctoral research was in the area of automated upgrading of older applications to use the new APIs of software components. For his research contributions, he was awarded the 1st Prize at ACM SIGPLAN Student Research Competition, held at OOPSLA 2005. In 2006, he obtained the 1st Prize at inter-disciplinary Grand Finals of ACM Student Research Competition, awarded during the ACM Turing Award banquet. In 2007, the Graduate College at the University of Illinois awarded him the Dissertation Completion Fellowship. He received his Ph.D. from the University of Illinois in 2007 with his dissertation entitled *Automated Upgrading of Component-Based Applications*.

After completing his Ph.D., Dr. Dig will work as a postdoctoral research associate at MIT. He will work on program transformations that retrofit parallelism into existing sequential applications.

Dr. Dig's goal in life is to learn and grow every day so that he can make a difference in the lives of people that interact with him. He delights in inspiring undergraduate and graduate students to find their passion for scientific research.